L Number	Hits	Search Text	DB	Time stamp
2	10	cache same USB same SRAM	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/17 12:42
3	300	cache same USB	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/17 12:53
4	34600	SRAM	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/17 12:42
5	43	(cache same USB) AND SRAM	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/17 12:43
6	101576	hit or miss	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/17 12:43
7	1141791	index or descriptor$3	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/17 12:44
8	19	((cache same USB) AND SRAM) and (hit or miss)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/17 12:44
9	15	(index or descriptor$3) and (((cache same USB) AND SRAM) and (hit or miss))	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/17 12:44
10	1921	cache and USB	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/17 12:53
11	334	(cache and USB) and (hit or miss)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/17 12:54

12	148	((cache and USB) and (hit or miss)) and (index or descriptor$3)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/17 12:54
13	44422	hierarch$3	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/17 12:54
14	49	(((cache and USB) and (hit or miss)) and (index or descriptor$3)) and hierarch$3	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/17 12:55

US-PAT-NO: 6405289

DOCUMENT-IDENTIFIER: US 6405289 B1

TITLE: Multiprocessor system in which a **cache** serving as a highest point of coherency is indicated by a snoop response

DATE-ISSUED: June 11, 2002

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Arimilli; Ravi Kumar	Austin	TX	N/A	N/A
Clark; Leo James	Georgetown	TX	N/A	N/A
Fields, Jr.; James Stephen	Austin	TX	N/A	N/A
Guthrie; Guy Lynn	Austin	TX	N/A	N/A

US-CL-CURRENT: 711/145, 711/119 , 711/122 , 711/141 , 711/144 , 711/146 , 711/151

ABSTRACT:

A method of maintaining **cache** coherency, by designating one **cache** that owns a line as a highest point of coherency (HPC) for a particular memory block, and sending a snoop response from the **cache** indicating that it is currently the HPC for the memory block and can service a request. The designation may be performed in response to a particular coherency state assigned to the **cache** line, or based on the setting of a coherency token bit for the **cache** line. The processing units may be grouped into clusters, while the memory is distributed using memory arrays associated with respective clusters. One memory array is designated as the lowest point of coherency (LPC) for the memory block (i.e., a fixed assignment) while the **cache** designated as the HPC is dynamic (i.e., changes as different caches gain ownership of the line). An acknowledgement snoop response is sent from the LPC memory array, and a combined response is

returned to the requesting device which gives priority to the HPC snoop response over the LPC snoop response.

14 Claims, 6 Drawing figures

Exemplary Claim Number: 1

Number of Drawing Sheets: 5

---------- KWIC ---------

Abstract Text - ABTX (1):
A method of maintaining **cache** coherency, by designating one **cache** that owns a line as a highest point of coherency (HPC) for a particular memory block, and sending a snoop response from the **cache** indicating that it is currently the HPC for the memory block and can service a request. The designation may be performed in response to a particular coherency state assigned to the **cache** line, or based on the setting of a coherency token bit for the **cache** line. The processing units may be grouped into clusters, while the memory is distributed using memory arrays associated with respective clusters. One memory array is designated as the lowest point of coherency (LPC) for the memory block (i.e., a fixed assignment) while the **cache** designated as the HPC is dynamic (i.e., changes as different caches gain ownership of the line). An acknowledgement snoop response is sent from the LPC memory array, and a combined response is returned to the requesting device which gives priority to the HPC snoop response over the LPC snoop response.

TITLE - TI (1):
Multiprocessor system in which a **cache** serving as a highest point of coherency is indicated by a snoop response

Brief Summary Text - BSTX (3):
The present invention generally relates to computer systems, specifically to a method of maintaining **cache** coherency in a multiprocessor computer system, and more particularly to an improved **cache** coherency protocol which allows the point of coherency (or coherency control) to dynamically float so as to more

efficiently handle **cache** operations in systems with complicated interconnection
topologies.

Brief Summary Text - BSTX (5):
The basic structure of a conventional multiprocessor computer system 10 is
shown in FIG. 1. Computer system 10 has several processing units, two of which
12a and 12b are depicted, which are connected to various peripheral devices,
including input/output (I/O) devices 14 (such as a display monitor, keyboard,
graphical pointer (mouse), and a permanent storage device or hard disk), memory
device 16 (such as random access memory or RAM) that is used by the processing
units to carry out program instructions, and firmware 18 whose primary purpose
is to seek out and load an operating system from one of the peripherals
(usually the permanent memory device) whenever the computer is first turned on.
Processing units 12a and 12b communicate with the peripheral devices by various
means, including a generalized interconnect or bus 20, or direct memory access
channels (not shown). Computer system 10 may have many additional components
which are not shown, such as serial, parallel, and universal system bus (**USB**)
ports for connection to, e.g., modems, printers or scanners. There are other
components that might be used in conjunction with those shown in the block
diagram of FIG. 1; for example, a display adapter might be used to control a
video display monitor, a memory controller can be used to access memory 16,
etc. The computer can also have more than two processing units.

Brief Summary Text - BSTX (6):
In a symmetric multiprocessor (SMP) computer, all of the processing units
are generally identical, that is, they all use a common set or subset of
instructions and protocols to operate, and generally have the same
architecture. A typical architecture is shown in FIG. 1. A processing unit
includes a processor core 22 having a plurality of registers and execution
units, which carry out program instructions in order to operate the computer.
An exemplary processing unit includes the PowerPC.TM. processor marketed by

International Business Machines Corp. The processing unit can also have one or more caches, such as an instruction **cache** 24 and a data **cache** 26, which are implemented using high speed memory devices. Caches are commonly used to temporarily store values that might be repeatedly accessed by a processor, in order to speed up processing by avoiding the additional latency of loading the values from memory 16. These caches are referred to as "on-board" when they are integrally packaged with the processor core on a single integrated chip 28. Each **cache** is associated with a **cache** controller (not shown) that manages the transfer of data and instructions between the processor core and the **cache** memory.

Brief Summary Text - BSTX (7):

A processing unit can include additional caches, such as **cache** 30, which is referred to as a level 2 (L2) **cache** since it supports the on-board (level 1) caches 24 and 26. In other words, **cache** 30 acts as an intermediary between memory 16 and the on-board caches, and can store a much larger amount of information (instructions and data) than the on-board caches can, but at a longer access penalty. For example, **cache** 30 may be a chip having a storage capacity of 512 kilobytes, while the processor may be an IBM PowerPC.TM. 604-series processor having on-board caches with 64 kilobytes of total storage. **Cache** 30 is connected to bus 20, and all loading of information from memory 16 into processor core 22 must come through **cache** 30. Although FIG. 1 depicts only a two-level **cache hierarchy,** multi-level **cache hierarchies** can be provided where there are many levels (L3, L4, etc.) of serially connected caches.

Brief Summary Text - BSTX (8):

In a multi-level **cache,** if a copy of a value is in every level of the **cache,** **the cache hierarchy** is referred to as being "inclusive." It is not necessary, however, to keep a copy of each value in the lower levels, and an inclusivity .

bit field may be added to the caches to indicate whether or not the **cache** is inclusive. For example, a three-level **cache** structure might provide an L3 **cache** which was not inclusive, such that a value residing in the L2 **cache** might not be present in the L3 **cache**. In this example, if an L2 **cache** issues a read command for a value that is not present in any of the caches of that processing unit, it can be passed to that L2 **cache** without (necessarily) loading it into the L3 **cache**.

Brief Summary Text - BSTX (9):

In an SMP computer, it is important to provide a coherent memory system, that is, to cause write operations to each individual memory location to be serialized in some order for all processors. By way of example, assume a location in memory is modified by a sequence of write operations to take on the values: 1, 2, 3, 4. In a **cache** coherent system, all processors will observe the writes to a given location to take place in the order shown. However, it is possible for a processing element to **miss** a write to the memory location. A given processing element reading the memory location could see the sequence 1, 3, 4, missing the update to the value 2. A system that implements these properties is said to be "coherent". Nearly all coherency protocols operate only to the granularity of the size of a **cache** block. That is to say, the coherency protocol controls the movement of and write permissions for operand data or instructions on a **cache** block basis, and not separately for each individual memory location.

Brief Summary Text - BSTX (10):

There are a number of protocols and techniques for achieving **cache** coherence that are known to those skilled in the art. All of these mechanisms for maintaining coherency require that the protocols allow only one processor to have a "permission" that allows a write operation to a given memory location (**cache** block) at any given point in time. As a consequence of this

requirement, whenever a processing element attempts to write to a memory
location, it must first inform all other processing elements of its desire to
write the location and receive permission from all other processing elements to
carry out the write.

Brief Summary Text - BSTX (11):
To implement **cache** coherency in a system, the processors communicate over a
common generalized interconnect (i.e., bus 20). The processors pass messages
over the interconnect indicating their desire to read from or write to memory
locations. When an operation is placed on the interconnect, all of the other
processors "snoop"(monitor) this operation and decide if the state of their
caches can allow the requested operation to proceed and, if so, under what
conditions. There are several bus transactions that require snooping and
follow-up action to honor the bus transactions and maintain memory coherency.
The snooping operation is triggered by the receipt of a qualified snoop
request, generated by the assertion of certain bus signals. Instruction
processing is interrupted only when a snoop **hit** occurs and the snoop state
machine determines that an additional **cache** snoop is required to resolve the
coherency of the offended sector.

Brief Summary Text - BSTX (12):
This communication is necessary because, in systems with caches, the most
recent valid copy of a given block of memory may have moved from the system
memory 16 to one or more of the caches in the system (as mentioned above). If
a processor (say 12a) attempts to access a memory location not present within
its **cache hierarchy**, the correct version of the block, which contains the
actual (current) value for the memory location, may either be in the system
memory 16 or in one of more of the caches in another processing unit, e.g.
processing unit 12b. If the correct version is in one or more of the other
caches in the system, it is necessary to obtain the correct value from the
cache(s) in the system instead of system memory.

Brief Summary Text - BSTX (13):

For example, consider a processor, say 12a, attempting to read a location in memory. It first polls its own L1 **cache** (24 or 26). If the block is not present in the L1 **cache**, the request is forwarded to the L2 **cache** (30). If the block is not present in the L2 **cache**, the request is forwarded on to lower **cache** levels, e.g., the L3 **cache**. If the block is not present in the lower level caches, the request is then presented on the generalized interconnect (20) to be serviced. Once an operation has been placed on the generalized interconnect, all other processing units snoop the operation and determine if the block is present in their caches. If a given processing unit has the block requested by processing unit in its L1 **cache**, and the value in that block is modified, and any lower level caches also have copies of the block, then their copies are stale, since the copy in the processor's **cache** is modified. Therefore, when the lowest level **cache** (e.g., L3) of the processing unit snoops the read operation, it will determine that the block requested is present and modified in a higher level **cache**. When this occurs with an in-line **cache** structure, the L3 **cache** places a message on the generalized interconnect informing the processing unit that it must "retry" it's operation again at a later time, because the actual value of the memory location is in the L1 **cache** at the top of the memory **hierarchy** and must be retrieved to make it available to service the read request of the initiating processing

Brief Summary Text - BSTX (14):

Once the request from an initiating processing unit has been retried, the L3 **cache** begins a process to retrieve the modified value from the L1 **cache** and make it available at the L3 **cache**, main memory or both, depending on the exact details of the implementation. To retrieve the block from the higher level caches, the L3 **cache** sends messages through the inter-**cache** connections to the higher level caches, requesting that the block be retrieved. These

messages
propagate up the processing unit **hierarchy** until they reach the L1 **cache** and
cause the block to be moved down the **hierarchy** to the lowest level (L3 or main
memory) to be able to service the request from the initiating processing unit.

Brief Summary Text - BSTX (15):
The initiating processing unit eventually re-presents the read request on
the generalized interconnect. At this point, however, the modified value has
been retrieved from the L1 **cache** of a processing unit and placed into system
memory, and the read request from the initiating processor will be satisfied.
The scenario just described is commonly referred to as a "snoop push". A read
request is snooped on the generalized interconnect which causes the processing
unit to "push" the block to the bottom of the **hierarchy** to satisfy the read
request made by the initiating processing unit.

Brief Summary Text - BSTX (16):
Thus, when a processor wishes to read or write a block, it must communicate
that desire with the other processing units in the system in order to maintain
cache coherence. To achieve this, the **cache** coherence protocol associates with
each block in each level of the **cache hierarchy**, a status indicator indicating
the current "state" of the block. The state information is used to allow
certain optimizations in the coherency protocol that reduce message traffic on
the generalized interconnect and the inter-**cache** connections. As one example
of this mechanism, when a processing unit executes a read it receives a message
indicating whether or not the read must be retried (i.e., reissued later). If
the read operation is not retried, the message usually also includes
information allowing the processing unit to determine if any other processing
unit also has a still active copy of the block (this is accomplished by having
the other lowest level caches give a "shared" or "not shared" indication for
any read they do not retry). Therefore, a processing unit can determine
whether any other processor in the system has a copy of the block. If

no other
processing unit has an active copy of the block, the reading processing unit
marks the state of the block as "exclusive". If a block is marked exclusive it
is permissible to allow the processing unit to later write to the block without
first communicating with other processing units in the system because no other
processing unit has a copy of the block. Therefore, it is possible for a
processor to read or write a location without first communicating this
intention onto the interconnection, but only where the coherency protocol has
ensured that no other processor has an interest in the block.

Brief Summary Text - BSTX (17):

The foregoing **cache** coherency technique is implemented in a specific protocol referred to as "MESI," and illustrated in FIG. 2. In this protocol, a
cache block can be in one of four states, "M" (Modified), "E" (Exclusive), "S"
(Shared) or "I" (Invalid). Under the MESI protocol, each **cache** entry (e.g., a
32-byte sector) has two additional bits which indicate the state of the entry,
out of the four possible states. Depending upon the initial state of the entry
and the type of access sought by the requesting processor, the state may be
changed, and a particular state is set for the entry in the requesting
processor's **cache**. For example, when a sector is in the Modified state, the
addressed sector is valid only in the **cache** having the modified sector, and the
modified value has not been written back to system memory. When a sector is
Exclusive, it is present only in the noted sector, and is consistent with
system memory. If a sector is Shared, it is valid in that **cache** and in at
least one other **cache**, all of the shared sectors being consistent with system
memory. Finally, when a sector is Invalid, it indicates that the addressed
sector is not resident in the **cache**. As seen in FIG. 2, if a sector is in any
of the Modified, Shared or Invalid states, it can move between the states
depending upon the particular bus transaction. While a sector in an Exclusive
state can move to any other state, a sector can only become Exclusive if it is
first Invalid.

Brief Summary Text - BSTX (18):

A further improvement in accessing **cache** blocks can be achieved using the **cache** coherency protocol. This improvement referred to as "intervention," allows a **cache** having control over a memory block to provide the data in that block directly to another **cache** requesting the value (for a read-type operation), in other words, bypassing the need to write the data to system memory and then have the requesting processor read it back again from memory. Intervention can generally be performed only by a **cache** having the value in a block whose state is Modified or Exclusive. In both of these states, there is only one **cache** block that has a valid copy of the value, so it is a simple matter to source (write) the value over the bus 20 without the necessity of first writing it to system memory. The intervention procedure thus speeds up processing by avoiding the longer process of writing to and reading from system memory (which actually involves three bus operations and two memory operations). This procedure not only results in better latency, but also increased bus bandwidth.

Brief Summary Text - BSTX (19):

There are many variations of the MESI protocol. The tagged ("T") state is used to identify a **cache** block which is inconsistent with system memory (i.e., modified) and is further responsible for writing the correct (current) value to memory upon deallocation (or to pass on the tag to another **cache** block during intervention). The T state can be used to share a modified value, by marking one of the sharing blocks as (temporarily) responsible for maintaining a valid copy of the value. The recently read ("R") state can be used to allow intervention when the value is unmodified but shared among many caches, so as to conveniently mark a single one of the sharing caches as being responsible for intervention. The hover ("H") state allows a **cache** line to maintain an address in the directory even though the corresponding value in the **cache** entry array is an invalid copy, so that it can snoop the correct value for its processing unit if the value happens to be broadcast as part of an

intervention
between the caches of two other processing units.

Brief Summary Text - BSTX (21):
Controller 32 is also typically responsible for protecting the target **cache** line until non-retry responses are received from all of the snoopers. For example, a bus master (processing unit or I/O device) may issue a read-with-intent-to-modify (RWITM) request, and one of the snoopers may provide a response indicating that its **cache** currently holds the requested value and can intervene it, but other snoopers may issue retry responses. In such a case, controller 32 ensures that no other master accesses the requested memory block until completion of the first operation.

Brief Summary Text - BSTX (22):
As the number of processing units in multiprocessor systems has increased, the interconnection topologies for the processing units, **cache** and system memory have become quite complicated, and can lead to inefficiencies in the handling of coherency operations. Multiprocessor systems with large numbers of processors (e.g., 32) are often grouped into processing clusters, with two or more processor sharing lower level (L2) caches, and the processing clusters being coupled to one another using multiple buses, switches and cross-bars. In such systems, all snooping devices may not receive a master's request at the same time due to different delays through the interconnection topology. The intricate nature of these connections makes it more difficult to collect all responses in a timely manner, and form a proper combined response, particularly when a device in one processing cluster contains a value that is requested by a device in a different processing cluster. The difficulty is compounded if two masters simultaneously issue conflicting access requests to different sectors of the same **cache** line. The result is an overall reduction in system performance.

Brief Summary Text - BSTX (24):
In light of the foregoing, it would be desirable to provide a method

of
serializing **cache** transactions which did not require a central point of coherency, such as that afforded by a system controller. It would be further
advantageous if the method could easily protect a requested **cache** line until
completion of an initial **cache** operation without adversely affecting system
performance.

Brief Summary Text - BSTX (26):
It is therefore one object of the present invention to provide an improved
method of maintaining **cache** coherency in a multiprocessor system.

Brief Summary Text - BSTX (28):
It is yet another object of the present invention to provide an improved
method of handling **cache** operations in a multiprocessor system having a plurality of processor clusters and a distributed system memory.

Brief Summary Text - BSTX (29):
The foregoing objects are achieved in a method of maintaining **cache** coherency in a multiprocessor computer system, comprising the steps of designating a single **cache** as a highest point of coherency (HPC) for a particular memory block (wherein the **cache** contains a valid value corresponding
to the memory block), issuing a request for the value from a master device of
the computer system, and sending a snoop response from the **cache** indicating
that it is currently the HPC for the memory block and can service the request.
The designation may be performed in response to a particular coherency state
assigned to the **cache** line which uniquely identifies the **cache** line with
respect to the memory block or, alternatively, based on the setting of a
coherency token bit for the **cache** line. In an illustrative embodiment, the
processing units are grouped into clusters which are coupled using cross-bar
switches, and the memory is distributed using memory arrays associated with
respective clusters. One of the memory arrays, which contains the memory
block, is further designated as a lowest point of coherency (LPC) for the
memory block. This memory array can first protect the memory block from any
conflicting operation, and the **cache** thereafter will protect the memory

block
until completion of the requested transaction. An acknowledgement snoop
response is sent from the LPC memory array and the HPC **cache**, and a combined
response is returned to the master device which gives priority to the snoop
response from the HPC **cache** over the acknowledgement snoop response from the
memory array, i.e., the HPC controls the coherency response. The combined
response may include an indication of a coherency state to be assigned to a
cache line of the master device upon receiving the value.

Drawing Description Text - DRTX (4):
FIG. 2 is a state diagram depicting a prior art **cache** coherency protocol
(MESI);

Detailed Description Text - DETX (2):
With reference now to the FIG., and in particular with reference to FIG. 3,
there is depicted one embodiment 40 of a multiprocessor computer system
constructed in accordance with the present invention. Computer system 40
comprises a plurality of processor clusters 42a, 42b, 42c and 42d, and a
distributed memory including mutually exclusive memory arrays 44a, 44b, 44c and
44d. Each memory array is associated with a respective processor cluster. The
processor clusters are coupled using four switches 46a, 46b, 46c and 46d.
Computer system 40 further includes a plurality of input/output (I/O) devices
which are not shown for simplicity, that may be accessed via an interconnect
bus connected to the switches. The present invention is directed to a method
of handling **cache** operations in a memory-coherent, multiprocessor system such
as the system of FIG. 4, but the present invention could be applied to computer
systems that have additional hardware components not shown in FIG. 4, or having
a different interconnection architecture (or both), so those skilled in the art
will appreciate that the present invention is not limited to the generalized
system shown in that figure.

Detailed Description Text - DETX (3):

An exemplary design for the processor clusters is further shown in FIG. 4.
In the illustrative embodiment, computer system 40 is a 32-processor system,
with eight processing units being provided in a given processor cluster.
Processor cluster 42a thus has eight processing units 48a, 48b, 48c, 48d, 48e,
48f, 48g and 48h. The processing units are paired such that a single lower
level (L2) **cache** is provided for a given pair. Processing units 48a and 48b
use L2 **cache** 50a; processing units 48c and 48d use L2 **cache** 50b; processing
units 48e and 48f use L2 **cache** 50c; and processing units 48g and 48h use L2
cache 50d. Inter-**cache** connections are provided by a local bus 52, and each of
the L2 caches is also coupled to switch 46a. Local bus 52 may be used to
access the associated memory array (44a). A similar construction is provided
for processor clusters 42b, 42c and 42d. Hereinafter, a given processor is
identified using a subscript notation P.sub.x.y, where the "x" **index** indicates
the particular processor cluster (0-3), and the "y" **index** indicates the
particular processing unit within the cluster (0-7). So, for example,
processing unit 48f may be referred to as processor P.sub.0.5.

Detailed Description Text - DETX (5):
One preferred method is to assign a "highest" point of coherency to any
cache line which can be uniquely identified as having a valid copy of the
requested value. Thus, a **cache** line in the modified (M) or exclusive (E)
states may be designated as the highest point of coherency (HPC) since such a
cache line is held to the exclusion of all other caches of other processing
units. Similarly, a **cache** line in the tagged (T) state may be designated the
HPC, even though other caches share the value, since the T state may be used to
uniquely identify a single **cache** line as being responsible for writing the
value to memory upon deallocation. The particular memory array (44a-44d)
which contains the target memory block is designated as the "lowest" point of
coherency (LPC). Only one LPC exists for a particular **cache** line, and its
assignment is fixed to a particular memory array.

Detailed Description Text - DETX (6):
According to this implementation, a **cache** line that is the HPC controls the coherency response (i.e., takes priority). Consider, for example, the situation wherein two processing units in different clusters (e.g., processors P.sub.0.0 and P.sub.1.0) simultaneously issue read-with-intent-to-modify (RWITM) requests for the same memory block (these operations could be asynchronous requests directed to two different sectors of the same block). A **cache** in a third processor cluster, which contains a current copy of the requested value, may not see the two requests in the same order as seen by the particular memory array which contains the target block. The **cache** containing the value (the HPC) will issue an acknowledgement signal (HPC_Ack) to the master that it first detects (say, P.sub.0.0) to indicate that it can service the request, and issue a retry signal (HPC_Retry) to the master that is later detected (P.sub.1.0). If the memory array (the LPC) sees the requests in the opposite order, it will issue an acknowledgement (LPC_Ack) to P.sub.1.0, and issue a retry (LPC_Retry) to P.sub.0.0. The combined response logic will receive the HPC_Ack signal and know to give priority to that signal over the LPC_Ack response. Therefore, processor P.sub.0.0 will proceed with its transaction first.

Detailed Description Text - DETX (7):
The combined response (a multi-bit signal) will indicate the state that should be assigned to the **cache** line for the master. The combined response may therefore be in the format of "HPC_go_X" or "LPC_go_X" where "X" is the state to be assigned, e.g., "go_S" for shared, "go_M" for modified, or "go_T" for tagged, etc. The combined response is formulated and sent by a controller at the master node (cluster), so it is seen first by the master and then by the snoopers. The combined response of "LPC_go_X" will be sent only if there are no retry snoop responses (in addition to no HPC response), since a retrying snooper may be the HPC (and LPC data may be stale). In other words, if no HPC

snoop response is received but a retry snoop response is received, then the
combined response will be a retry.

Detailed Description Text - DETX (8):
An HPC may or may not exist at any given time for a particular **cache** line.
In the absence of any HPC response, the LPC memory array will serialize the
transactions. Any memory array which does not own the target memory block
simply issues a null response. After all transactions are completed, the new
owner of the **cache** block becomes the new point of coherency for that block.

Detailed Description Text - DETX (11):
In an alternative implementation, each **cache** line is provided with an extra
bit that is used as a point of coherency token. If the bit is set (i.e.,
"high" or "on"), then that **cache** line is designated the HPC, regardless of its
coherency state. In this manner, a **cache** line may become the HPC even if it is
not in the M or T state. This variation is particularly useful when the T
state may not be implemented as a fully unique state, such as in a hierarchical
implementation wherein the T state is used locally within a processor cluster
as opposed to a global indication for caches in all processor clusters. The
HPC bit may be set by the operating system, or via a special instruction set
architecture (such as an extension of the PowerPC.TM. instruction set
architecture) which provides instructions (e.g., load and store instructions)
having a special flag to indicate whether the bit should be set.

Detailed Description Text - DETX (12):
For multi-level **cache hierarchies,** any **cache** level may be designated an HPC.
Although FIG. 4 illustrates only a two-**cache hierarchy,** it is understood that
the present invention could be applied as well to, e.g., an L3 **cache**.

Claims Text - CLTX (1):
1. A method of maintaining **cache** coherency in a multiprocessor computer
system including a plurality of memory arrays and a plurality of processing

units that each have at least one **cache**, said method comprising:

Claims Text - CLTX (3):
designating a single **cache** containing a valid value corresponding to the memory block as a highest point of coherency for the memory block;

Claims Text - CLTX (7):
the **cache** designated as the highest point of coherency sending a snoop response indicating that the **cache** is currently the highest point of coherency for the memoryblock and can service the request and thereafter protecting the memory block from any conflicting operation until completion of a transaction associated with the request.

Claims Text - CLTX (8):
2. The method of claim 1 wherein designating a single **cache** includes assigning to a **cache** line in the **cache** a coherency state which uniquely identifies the **cache** line with respect to the memory block.

Claims Text - CLTX (9):
3. The method of claim 1 wherein designating a single **cache** includes the step of setting a coherency token bit for a **cache** line in the **cache**.

Claims Text - CLTX (10):
4. The method of claim 1 wherein each processing unit has a multi-level **cache hierarchy**, and designating a single **cache** comprises designating a second level **cache** as the highest point of coherency for the memory block.

Claims Text - CLTX (14):
returning a combined response to the master device, in response to said sending steps, wherein the combined response gives priority to the snoop response from the **cache** over the acknowledgement snoop response from the memory array.

Claims Text - CLTX (15):
7. The method of claim 6 wherein said returning step returns the combined

response with an indication of a coherency state to be assigned to a **cache** line
of the master device upon receiving the value.

Claims Text - CLTX (18):
a plurality of processing units each having a **cache**;

Claims Text - CLTX (20):
cache coherency means for (1) designating a memory array containing a memory
block as a lowest point of coherency for the memory block, (2) designating a
single one of said caches containing a valid value corresponding to the memory
block as a highest point of coherency for the memory block, and (3) sending a
snoop response indicating that said **cache** is currently the highest point of
coherency for the memory block and can service a request for the value issued
from a master device;

Claims Text - CLTX (21):
wherein said memory array designated as the lowest point of coherency first
protects the memory block from any conflicting operation and said **cache**
designated as the highest point of coherency thereafter protects the memory
block from any conflicting operation until completion of a transaction
associated with the request.

Claims Text - CLTX (22):
9. The computer system of claim 8 wherein said **cache** coherency means
designates said **cache** as the highest point of coherency based on a coherency
state assigned to a **cache** line of said **cache** which uniquely identifies said
cache line with respect to the memory block.

Claims Text - CLTX (23):
10. The computer system of claim 8 wherein said **cache** coherency means
designates said **cache** as the highest point of coherency based on a setting of a
coherency token bit for a **cache** line in said **cache**.

Claims Text - CLTX (25):
each processing unit has a multi-level **cache hierarchy**; and

Claims Text - CLTX (26):
said **cache** is a second level **cache**.

Claims Text - CLTX (28):
13. The computer system of claim 8 wherein said **cache** coherency means
further sends an acknowledgement snoop response from said memory array designated as the lowest point of coherency, and returns a combined response to
said master device giving priority to the snoop response from said **cache** over
the acknowledgement snoop response from said memory array.

Claims Text - CLTX (29):
14. The computer system of claim 13 wherein said **cache** coherency means
returns the combined response with an indication of a coherency state to be
assigned to a **cache** line of said master device upon receiving the value.

Other Reference Publication - OREF (1):
Jim Handy, "The **Cache** Memory Book", Academic Press, Inc., pp 48-61, 87-91,
and 158-190, 1993.

US-PAT-NO: 6622208

DOCUMENT-IDENTIFIER: US 6622208 B2

TITLE: System and methods using a system-on-a-chip with soft
cache

DATE-ISSUED: September 16, 2003

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
North; Gregory Allen	Austin	TX	N/A	N/A

US-CL-CURRENT: 711/118, 711/128 , 711/202 , 711/203 , 711/212 , 711/220
, 711/3

ABSTRACT:

A soft cache system compares tag bits of a virtual address with tag fields
of a plurality of soft cache register entries, each entry associated with an
index to a corresponding cache line in virtual memory. A cache line size for
the cache line is programmable. When the tag bits of the virtual address match
the tag field of one of the soft cache entries, the **index** from that entry is
selected for generating a physical address. The physical address is generated
using the selected **index** as an offset to a corresponding soft cache space in
memory.

18 Claims, 14 Drawing figures

Exemplary Claim Number: 1

Number of Drawing Sheets: 8

---------- KWIC ---------

Abstract Text - ABTX (1):
A soft cache system compares tag bits of a virtual address with tag fields
of a plurality of soft cache register entries, each entry associated with an

index to a corresponding cache line in virtual memory. A cache line size for the cache line is programmable. When the tag bits of the virtual address match the tag field of one of the soft cache entries, the **index** from that entry is selected for generating a physical address. The physical address is generated using the selected **index** as an offset to a corresponding soft cache space in memory.

Brief Summary Text - BSTX (8):

According to the inventive concepts, a cache system is disclosed which compares tag bits of a virtual address with tag fields of a plurality of soft cache register entries, each entry associated with an **index** to a corresponding cache line in a programmable cache memory space. A cache line size of the cache line is also programmable. When the tag bits of the virtual address match the tag field of one of the soft cache entries, the **index** from that entry is selected for generating a physical address. The physical address is generated using the selected **index** as an offset to a corresponding programmable soft cache space in memory address by block address bits from the virtual address.

Brief Summary Text - BSTX (9):

Systems and software embodying the principles of the present invention advantageously allow software caching in systems with limited or no hardware caching support. An example is the disclosed system-on-a-chip employing an ARM7tdmi CPU core, which has neither a hardware cache nor a memory management unit. Additionally, since cache line replacement is done in software on a cache **miss**, the interface between memories operating under different protocols has increased efficiency.

Drawing Description Text - DRTX (8):

FIG. 6 is a functional block diagram of the external **SRAM**/Flash memory controller (SMC) of the system of FIG. 1;

Detailed Description Text - DETX (7):
Among the other devices operating directly off main AHB bus 103 are a 4-channel DMA engine 106, and flash/**SRAM** interface 107, including an external memory controller, which maps up to 512 MByte external memory into the microprocessor memory space as an extension of on-chip memory, a test interface controller (TIC) 108, arbiter 109 and LCD interface 110. Test Interface Controller (TIC) 108 can take over the bus control from microprocessor 101 and mimic the bus cycle in order to stimulate the blocks connected to AHB/APB buses. Arbiter 109 arbitrates bus requests on main bus 103. LCD interface 110 supports connections to various LCD panels (since the display may require a large frame buffer, display controller 110 operates from the high speed bus).

Detailed Description Text - DETX (50):
External **SRAM**/Flash Memory Controller (SMC) 107 supports eight external memory blocks, each having an address space up to 6 M Bytes. In the preferred embodiment of system 100, 4 SRAMs blocks are used, each with 1 M-byte address space, along with one block of FLASH RAM, as shown in FIG. 6. Each Bank has its own configuration register with which programmers can configure the Bank to support a specific type of External memory. In FIG. 6, the banks designated BANK 0-3 comprise 16-bit external SRAMs and the bank designated BANK4 comprises 8-bit wide external Flash RAMs.

Detailed Description Text - DETX (65):
USB port 114 can operate in either Configuration 0, where the control endpoint is for standard commands or Configuration 1, where the control endpoint is for Vendor/Class commands. String **Descriptor** Control Logic 1110 decodes string commands in the standard configuration. Vendor/Class commands are not decoded, but instead are stored and an interrupt to microprocessor 101 by Vendor/Class Control Logic. Subsequently, microprocessor 101, under software control, decodes Vendor/Class commands.

Detailed Description Text - DETX (68):
Each of the debug sub-block has a 1 bit read only "owner" field.

When this
bit is "0" the owner of this block is Microprocessor 101 and when this bit is
"1" the owner is DSP 102. The owner can write to the "Other Wr" field which,
if set, allows the other processor to write to the remaining fields of the
debug block. The "Debug En" field is set before the actions of the debug block
become effective and is also used to save power when this block is not in use.
The "Clk Freeze En" is a 1-bit field which when set at the trigger of a debug
event freezes the DSP clock. "Mem" is a 2 bit field which selects memories to
be observed for debug event. Start address range and end address range are
set, so that when the address is between these ranges an event triggers. The
"**Hit** Count" tracks the number of times a debug event has occurred.
Num_hits_for_trigger is the field which sets the number of hits to the address
range before the interrupt is enabled or clock is frozen.

Detailed Description Text - DETX (83):
Historically, embedded DSPs and uncached CPUs have either run code directly
out of external parallel memory (DRAM, **SRAM**, NOR FLASH) or out of internal
SRAM/FLASH. With respects to running code from external parallel memory, CPU
speed and power are adversely impacted. On the other hand, internal **SRAM**/FLASH
memory is typically significantly smaller thereby forcing direct memory
management during code development to insure proper paging. In particular, all
routines must fit within certain footprint guidelines to allow paging and
concurrent operation. In certain applications where all of the code is
developed in assembly language, this restriction can be addressed; however,
using a higher level languages like "C" for fast code development, the problem
is difficult, if not impossible, to solve.

Detailed Description Text - DETX (85):
As discussed further below in detail, the present soft cache system solves
these problems for low-performance code. Among other things, a hardware assist
is provided to manage existing generic on-chip **SRAM** as a cache. Additionally,
the combination of less than 1000 gates and a software abort handler provide a

means of managing code residing in NAND Flash as directly executable. Moreover, this cache mechanism is optimized for slow rate memories with lines
size programmable at 128, 256, 512, and 1kbytes per line and uses 16 way set
associative logic to provide 16 lines of cache space within the local **SRAM**.

Detailed Description Text - DETX (86):
The soft cache system preferably uses external **SRAM** for storing code and
on-chip memory for data constants or other secure information. In the illustrated embodiment, a virtual (soft cache) memory space of 2 Mbytes of
external and internal **SRAM** is dedicated to soft cache, although the size, as
well as the location in memory, are not critical and can change in actual
implementations.

Detailed Description Text - DETX (90):
The cache tag from the virtual address is compared with the tag fields in
the corresponding 16 register entries by the hardware comparators at Step 1403.
If a match occurs, the **index** corresponding to the matching entry in soft cache
is taken for use in generating a physical address to the appropriate location
in the soft cache memory block. For a 16-way cache system, the entries are
indexed with four bits from 0 to 15. In an embodiment where the soft cache
address space starts at 0000 and works upward, bits N-1:0 of the CPU address
become bits N-1:0 of the physical address and address the location in the cache
line. The 4-bit **index** from the matching soft cache entry replaces bits N+3:N
of the physical address and operates as an offset which selects the proper
cache line. The remaining bits (31:N+4) come from the virtual address and are
used for RAM bank, block and chip select. The required read or write to the
addressed area in the soft cache memory space subsequently takes place.

Detailed Description Text - DETX (91):
For example, assume that the cache line size is programmed to be 256 bytes.
In this case, N=8. Bits 21:8 of the CPU address are therefore compared with
the cache tags in the table entries. Also assume that a **hit** occurs to

entry #5
and the corresponding 4-bit **index** is 5. For simplicity, the soft cacheable
block of memory will have an arbitrarily selected starting address of 0. The
physical address (in hex) is therefore 000005CC, where CC are bits 7:0 of the
CPU address to a location within cache line 5.

Detailed Description Text - DETX (93):
Next consider the case where a cache **miss** occurs during comparison Step
1403. A hardware soft cache history register is maintained which tracks the
last four hits (matches). Preferably, this register is a shift register which
shifts in the 4-bit **index** from each matching entry, if that **index** differs from
the **index** of the previous matching **index**. In other words, if two consecutive
indices are the same, no shifting takes place such that no two consecutive
indices in the shift register are equal. This register improves the
performance of the software replacement handler invoked on a cache **miss**.

Detailed Description Text - DETX (94):
On the **miss,** a soft cache abort is signaled to microprocessor 101 by setting
an Abort Status bit in register. (This bit is cleared automatically after
being read). The address causing the cache **miss** (abort) is written into an
Abort Address register. Setting the Abort Status bit initiates the replacement
handler routine.

Detailed Description Text - DETX (95):
The replacement handler routine selects the cache line to be replaced based
on the contents of the history register. For example, the cache line replaced
can be one of those not corresponding to the four indices stored in the history
register. The required data is fetched from its current location in memory,
using the address in the Abort Address register and loaded into the selected
cache line, and the cache tag in the corresponding cache entry is updated. The
source location could be in NAND or serial flash and the software handler is
capable of performing the specific access procedures normally required

for
these types of memory. To implement this memory, the DMA engine and other
system resources are invoked as required. Once the data has been encached and
the cache tag updated, the instruction causing the cache **miss** can be
successfully reissued.

Detailed Description Text - DETX (98):
One exemplary use of present soft **cache** is in digital audio player
applications of system 100. For example, all low-performance code can run
directly thru the soft **cache** with only the high performance code (**USB**,
Audio

Claims Text - CLTX (1):
1. A soft cache system performing the operations of: comparing tag bits of
a virtual address with tag bits programmed in a plurality of register entries,
each entry associated with an **index** to a cache line of programmable cache line
size in virtual cache space, the virtual cache space location in memory space
programmed by a block address portion of the virtual address and the cache line
size programmed by a field in the virtual address; when the tag bits of the
virtual address match the tag bits of one of the programmable register entries,
selecting the **index** corresponding to said entry for generating a physical
address; and generating the physical address using the selected **index** as an
offset to the cache line and the block address from the virtual memory as an
address to the virtual cache space; and when the tag bits of the virtual
address do not match the tag bits of one of the register entries: selecting a
cache line in the virtual cache space for encaching data associated with the
virtual address; retrieving the data associated with the virtual address from
a second area of memory using the virtual address; encaching the data
retrieved from the second area of memory in the selected cache line in the
virtual cache space; and updating the tag bits in the register associated with
the selected cache line.

Claims Text - CLTX (7):
7. A method of data caching comprising the steps of: setting up an

M-way
set associative cache comprising a register having M number of corresponding
entries each storing a tag value and indexed to a cache line in a selected
cache area in memory; setting a cache line size for the cache lines, the cache
tag value selected as a function of the selected cache line size; generating a
virtual address including a tag field, a cache line address field addressing
locations within the cache line size and a block address field addressing the
cache area in memory; comparing the tag field of the virtual address with the
tag values in the cache; when the tag field of the virtual address matches a
tag value in the cache, generating a physical address to the corresponding
cache line with the **index** associated with the register entry containing the
matching tag value and the cache line and block address fields from the virtual
address; updating a history register with the **index** associated with the
matching tag value; and when the tag field of the virtual address does not
match a tag value in the register, accessing a corresponding area in memory
with the virtual address; selecting a replacement cache line from the cache
area in memory from information stored in the history register; transferring
data from the corresponding area in memory accessed with the virtual address to
the replacement cache line in the cache area of memory; and updating the
register entry associated with the **index** to the replacement cache line with the
tag from the virtual address.

Claims Text - CLTX (8):
8. The method of claim 7 wherein said step of updating the history register
comprises the step of shifting an **index** into a shift register.

Claims Text - CLTX (14):
14. A system comprising: a central processing unit; a memory including a
soft cache space; and a soft cache system for accessing data in the soft cache
space at a programmable location within memory space in response to addresses
from the central processing unit including: a register having a

plurality of
entries each for storing a cache tag and indexed to a cache line in the soft
cache space, the cache line having a programmable cache line size; a plurality
of comparators for comparing a tag field of an address from the central
processing unit with cache tags stored in each of the entries of the register;
circuitry for generating an address to the soft cache space when the tag field
matches a cache tag in the register using the **index** associated with the entry
storing the matching cache tag and selected bits from the address from the
central processing unit addressing the soft cache space; and circuitry for
updating data within the soft cache space when the tag field does not match the
cache tag in the register and operable to select a cache line in the soft cache
space for encaching data associated with the address from the central
processing unit, retrieve the data from a second area of the memory space using
the address from the central processing unit, encache the data from the second
area of the memory space in the selected cache line, and update the cache tag
in the register associated with the selected cache line.

US-PAT-NO: 6499076

DOCUMENT-IDENTIFIER: US 6499076 B2

TITLE: Memory management for use with burst mode

DATE-ISSUED: December 24, 2002

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP	CODE	COUNTRY
Date; Atsushi	Tokyo	N/A	N/A		JP
Kato; Katsunori	Kawasaki	N/A	N/A		JP
Yokoyama; Noboru	Tokyo	N/A	N/A		JP
Maeda; Tadaaki	Kawasaki	N/A	N/A		JP
Fujiwara; Takafumi	Yokohama	N/A	N/A		JP

US-CL-CURRENT: 710/113, 710/110 , 711/113 , 711/119 , 711/147

ABSTRACT:

A P bus from a CPU, an MC bus from a system memory, an IO bus to which an input/output device has been connected, and a G bus for transferring image data of a scanner/printer controller are connected to a system bus bridge (SBB). The SBB connects any of the P bus, G bus and IO bus as a master and any of the MC bus and IO bus as a slave in dependence upon a request from a master. At this time the P bus and IO bus can be connected in parallel with the G bus and MC bus. As a result, access to the memory by the scanner/printer controller can be carried out in parallel with use of the input/output device by the CPU. This makes it possible to process a large quantity of data, such as image data, efficiently.

7 Claims, 67 Drawing figures

Exemplary Claim Number: 7

Number of Drawing Sheets: 60

---------- KWIC ---------

Brief Summary Text - BSTX (10):

Furthermore, a cache memory is used in the prior art to process data efficiently. Conventional cache control, however, is such that the cache is turned on and off based upon address information of the memory that is the destination of the data transfer. When a large quantity of data is transferred to a memory space for cache storage, therefore, a large quantity of data is cached and the memory space is rewritten entirely by new data. If another device accesses the memory, there is a good possibility of a cache **miss**. Though increasing cache storage capacity may appear to be a solution, this leads to a major increase in manufacturing cost. In particular, when printing or the like is carried out, a large quantity of data that has been read out is delivered to the printer engine and, even though the data has been cached, it is not used twice. Caching data indiscriminately in this manner rather lowers the cache **hit** rate. Thus, a fourth problem is that cache memory cannot be used efficiently.

Drawing Description Text - DRTX (18):

FIG. 16 is a timing chart showing the timing of single readout in a case where a **hit** has occurred in a memory front cache;

Drawing Description Text - DRTX (19):

FIG. 17 is a timing chart showing the timing of single readout in a case where a **hit** has not occurred in a memory front cache;

Drawing Description Text - DRTX (20):

FIG. 18 is a timing chart showing the timing of single write in a case where a **hit** has occurred in a memory front cache;

Drawing Description Text - DRTX (21):

FIG. 19 is a timing chart showing the timing of single write in a case where a **hit** has not occurred in a memory front cache;

Detailed Description Text - DETX (29):

FIG. 7 is a block diagram of the memory controller 403. The memory controller 403, which is connected to the MC bus, namely the special-purpose local bus of the memory controller, supports a synchronous DRAM (SDRAM) of a maximum of 1 GB and 32-MB flash ROM or ROM. In order to exploit the characteristic high speed of the SDRAM at the time of burst transfer, 64 (16.times.4)-burst transfer is implemented. Taking into account single transfer of continuous addresses from the CPU or IO bus, an **SRAM** (memory front cache) 702 is incorporated within the main controller and direct single transfer to the SDRAM is avoided to the maximum extent to thereby raise the transfer efficiency. The data bus width between the memory controller and the SDRAM is 72 bits for the signals ramData and ramPar (of which the 8-bit signal ramPar is parity), and the width of the data buses fntromData, prgromData between the memory controller and the flash ROM is 32 bits.

Detailed Description Text - DETX (75):

<**SRAM** control (memory front cache)>

Detailed Description Text - DETX (76):

An SDRAM used as a main memory provides very high burst transfer speed but such high speed cannot be achieved in case of a single transfer. Accordingly, a memory front cache is packaged within the memory controller to speed up single transfer. The memory front cache is composed of a cache controller 706 and an **SRAM** 702. Since the transfer master and transfer length can be ascertained by the mTType[6:0] signal defined for the MC bus, cache ON/OFF can be set for every master or for every transfer length. The cache schemes are as set forth below. It should be noted below that, unless stated otherwise, the term "simple cache" or "cache memory" refers not to a cache incorporated within the processor core but to a memory front cache incorporated in the main controller. 2-way set associative 8-KB data RAM 128.times.21.times.2 Tag RAM LRU (Least Recently Used) algorithm Write-thru No write allocate

Detailed Description Text - DETX (79):

If address Imaddr[31:0] is applied in case of single transfer (cache ON), then it is applied to b1_tag_ram 801, b2_tag_ram 802, b1_data_ram 702-a,

b2_data_ram702-b and lru 803 with lmaddr[11:5] serving as the **index,** and valid
bit "v" and b1_tag_addr"; valid bit "v" and b2_tag_addr; b1_out_data; b2_out_data; and lru_in, which correspond to the entered **index,** are output from
the respective blocks (step S902).

Detailed Description Text - DETX (80):
Next, b1_tag_addr and b2_tag_addr output by b1_tag_ram 801 and b2_tag_ram
802 are compared with the address lmaddr[31:12] by a b1_comparator 804 and
b2_comparator 805. The result, namely **hit or miss,** is reported to the cache
controller 706 by b1_**hit miss**_L, b2_**hit**_missL signals, whereby **hit or miss** is
judged (step S903).

Detailed Description Text - DETX (81):
In case of a **hit,** read or write is determined (step S904). If a **hit** is
detected, this is a case where the address lmaddr [31:12] agrees with either
b1_tag_addr or b2_tag_addr. If a **hit** is detected and then read is determined,
operation is as follows: If b1 is a **hit** and the requested transfer is read,
b1_out_data is selected of b1_out_data and b2_out_data that have already been
read out, and 8-byte data indicated by lmaddr[4:3] is output to the MC bus
(step S905). At the same time, lru corresponding to this **index** is rewritten as
"0" (=b1 **hit**) and the transfer is terminated. If b2 is a **hit** and the requested
transfer is read, b2_out_data is selected of b1_out_data and b2_out_data that
have already been read out, and 8-byte data indicated by lmaddr[4:3] is output
to the MC bus (step S905). At the same time, lru corresponding to this **index**
is rewritten as "1"h (=b2 **hit**) and the transfer is terminated.

Detailed Description Text - DETX (82):
On the other hand, if a **hit** is detected and then write is determined,
operation is as follows: If b1 is a **hit** and the requested transfer is write,
then, of the 8-byte data indicated by lmaddr[4.3] of b1_data_ram 702-a
indicated by the **index,** only a valid byte lane indicated by mBE_L[7.0] is
rewritten. At the same time, lru corresponding to this **index** is

rewritten as
"0"h (=b1 **hit**). Further, the SDRAM also is rewritten and transfer is terminated in similar fashion (step S906). If b2 is a **hit** and the requested
transfer is write, then, of the 8-byte data indicated by lmaddr[4.3] of b2_data_ram 702-b indicated by the **index**, only a valid byte lane indicated by
mBE_L[7.0] is rewritten. At the same time, lru corresponding to this **index** is
rewritten as "1"h (=b2 **hit**). Further, the SDRAM also is rewritten and transfer
is terminated in similar fashion (step S906).

Detailed Description Text - DETX (83):
If b1 and b2 are both misses, on the other hand, a read or write judgment is
rendered (step S1001). If the requested transfer is read, 8-byte data indicated by lmaddr[31:3] is read out of the SDRAM (step S1003) and is output
to the MC bus (step S1004). At the same time, lru corresponding to this **index**
is read out. If lru is "0"h, data from the SDRAM is written to b2_data_ram and
lru also is rewritten as "1"h. If lru is "1"h, data from the SDRAM is written
to b1_data_ram and lru also is rewritten as "0"h (step S1005), after which the
transfer is terminated. If b1 and b2 are both misses and the requested transfer is write, the data is merely written to the SDRAM and the transfer is
terminated (step S1002).

Detailed Description Text - DETX (93):
FIG. 16 illustrates the timing of single readout in case of a **hit** in the
memory front cache. Here b1/b2_out_data, which has been read out of the cache
memory b1_data_ram 702-a or b2_data_ram 702-b, is output as data mDataIn[63:0]
to be read out. The burst length of the SDRAM is 4 and the CAS latency is 3.
This corresponds to the processing at step S905 in FIG. 9.

Detailed Description Text - DETX (94):
FIG. 17 illustrates the timing of single readout in a case where there is no
hit in the memory front cache. Data ramData[63:0], which has been read out of
the SDRAM, is output as data mDataIn[63:0] to be read out. This data is
written also to cache memory b1_data_ram 702-a or b2_data_ram 702-b as b1/b2_in_data. The burst length of the SDRAM is 4 and the CAS latency is 3.

This corresponds to the processing at steps S1004 and S1005 in FIG. 10.

Detailed Description Text - DETX (95):
FIG. 18 illustrates the timing of single write in case of a **hit** in the memory front cache. Data mDataOut[63:0] to be written is written to cache memory b1_data_ram 702-a or b2_data_ram 702-b and to the SDRAM as well. The burst length of the SDRAM is 4 and the CAS latency is 3. This corresponds to the processing at step S906 in FIG. 9.

Detailed Description Text - DETX (96):
FIG. 19 illustrates the timing of single write in a case where there is no **hit** in the memory front cache. Data mDataOut[63:0] to be written is written only to the SDRAM and not to cache memory b1_data_ram 702-a or b2_data_ram 702-b. The burst length of the SDRAM is 4 and the CAS latency is 3. This corresponds to the processing at step S1002 in FIG. 10.

Detailed Description Text - DETX (111):
Further, read prefetch queuing of the external master is executed. As a result, continuous readout of a data stream from an external master is speeded up. Invalidation of the read buffer is performed 1. when new reading of the IO bus has not produced **hit** in the buffer; 2. when write from the CPU to the memory has been performed; 3. when write from the G bus to the memory has been performed; and 4. when write from the IO bus to the memory has been performed.

Detailed Description Text - DETX (221):
The G-bus write cycle is as follows: 1 The master issues a bus request and asserts gReq_L. 2 The arbiter grants permission, asserts gGnt_L and negates gReq_L. 3 The master receives gGnt_L and drives gTs_L, gAddr, gRdNotWr, gBstCnt. In case of a write operation, the master also drives gData simultaneously if gSlvBsy_L has not been asserted. If gSlvBsy_L is being driven, the master performs drive upon waiting for gSlvBsy_L to be freed. 4 The slave decodes the address when gTs_L has been asserted. If is **hit** occurs, i.e., if a decoded address is the device's own address, the device

recognizes
the transfer to itself. If gSlvBsy_L has not been asserted by another slave at
this time, drive of gSlvBsy_L and gAack is started. In case of a read operation, gData is driven as well. If gSlvBsy_L has been asserted by another
slave, this means that the data bus is currently in use. Drive is started,
therefore, upon waiting for this signal to be negated. If the slave can make
preparations for data transfer following the start of drive of gSlvBsy_L,
gAack_L, (gData), then each of these signals is asserted and data transfer is
begun. 5 The address phase ends and the master negates gAddr, gRdNotWr,
gBstCnt at the moment gAack_L is asserted. At this time, moreover, the master
changes over the write data every clock and transfers only the amount of data
specified by gBstCnt. The master and slave must become aware of the end of
data transfer by counting the clock themselves.

Detailed Description Paragraph Table - DETL (1):
TABLE 1 ITEM SUMMARY SPECIFICATIONS CHIP OPERATION INTERNAL: 100 MHz;
FREQUENCY EXTERNAL BUS & MEMORY BUS: 100 MHz PACKAGE 313-PIN BGA EXTERNAL
PCI INTERFACE MEMORY BUS VIDEO GENERAL-PURPOSE I/O IEEE1284 RS232C (**USB**)
LAN 100/10baseT LCP PANEL & KEYS INTERNAL CPU CORE BLOCKS PRIMARY **CACHE** MMU
ICU SYSTEM BUS BRIDGE CONTROLLER W. **CACHE** COPY ENGINE PLL POWER CONTROL
UNIT IO BUS ARBITER GRAPHICS BUS ARBITER DMA NUMBER FIVE CHANNELS
CONTROLLER OF CHANNELS MAX. 200 MB/s @ 50 MHz TRANSFER SPEED (PEAK)
TRANSFER- INTERNAL OUTPUT CAPABLE BLOCK .rarw..fwdarw.LOCAL MEMORY PATH
MEMORY & SUPPORT SDRAM BUS CONTROL MEMORY DATA 64 BITS WIDTH MAXIMUM 1 GB
MEMORY CAPACITY MAXIMUM 682 MB/s MEMORY BUS TRANSFER SPEED GRAPHICS BUS
ARBITRATION PRIORITY FIRST-COME FIRST- METHOD SERVED PROCESSING MAXIMUM 800
MB/s BUS TRANSFER SPEED BUS WIDTH 64 BITS. 100 MHz PCI BUS PCI BUS Rev
2.1, 32-BIT, 33M PCI FORMAT TRANSFER READ 96 MB/s, WRITE 88 SPEED MB/s WHEN
MASTER TRANSFER READ 101 EM/s, WRITE 111 SPEED MB/s PRIORITY FIRST-COME WHEN
FIRST- SLAVE IO BUS ARBITRATION SERVED PROCESSING METHOD 200 MB/s MAXIMUM
32 BITS, 50 MHz BUS TRANSFER SPEED BUS WIDTH

US-PAT-NO: 6484241

DOCUMENT-IDENTIFIER: US 6484241 B2

TITLE: Multiprocessor computer system with sectored **cache** line system bus protocol mechanism

DATE-ISSUED: November 19, 2002

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Arimilli; Ravi Kumar	Austin	TX	N/A	N/A
Dodson; John Steven	Pflugerville	TX	N/A	N/A
Guthrie; Guy Lynn	Austin	TX	N/A	N/A

US-CL-CURRENT: 711/141, 711/144 , 711/146

ABSTRACT:

A method of maintaining coherency in a multiprocessor computer system wherein each processing unit's **cache** has sectored **cache** lines. A first **cache** coherency state is assigned to one of the sectors of a particular **cache** line, and a second **cache** coherency state, different from the first **cache** coherency state, is assigned to the overall **cache** line while maintaining the first **cache** coherency state for the first sector. The first **cache** coherency state may provide an indication that the first sector contains a valid value which is not shared with any other **cache** (i.e., an exclusive or modified state), and the second **cache** coherency state may provide an indication that at least one of the sectors in the **cache** line contains a valid value which is shared with at least one other **cache** (a shared, recently-read, or tagged state). Other coherency states may be applied to other sectors in the same **cache** line. Partial intervention may be achieved by issuing a request to retrieve an entire **cache** line, and sourcing only a first sector of the **cache** line in response to the request. A second sector of the same **cache** line may be sourced from a

third
cache. Other sectors may also be sourced from a system memory device of the
computer system as well. Appropriate system bus codes are utilized to transmit
cache operations to the system bus and indicate which sectors of the **cache** line
are targets of the **cache** operation.

14 Claims, 4 Drawing figures

Exemplary Claim Number: 8

Number of Drawing Sheets: 4

---------- KWIC ---------

Abstract Text - ABTX (1):
A method of maintaining coherency in a multiprocessor computer system
wherein each processing unit's **cache** has sectored **cache** lines. A first **cache**
coherency state is assigned to one of the sectors of a particular **cache** line,
and a second **cache** coherency state, different from the first **cache** coherency
state, is assigned to the overall **cache** line while maintaining the first **cache**
coherency state for the first sector. The first **cache** coherency state may
provide an indication that the first sector contains a valid value which is not
shared with any other **cache** (i.e., an exclusive or modified state), and the
second **cache** coherency state may provide an indication that at least one of the
sectors in the **cache** line contains a valid value which is shared with at least
one other **cache** (a shared, recently-read, or tagged state). Other coherency
states may be applied to other sectors in the same **cache** line. Partial
intervention may be achieved by issuing a request to retrieve an entire **cache**
line, and sourcing only a first sector of the **cache** line in response to the
request. A second sector of the same **cache** line may be sourced from a third
cache. Other sectors may also be sourced from a system memory device of the
computer system as well. Appropriate system bus codes are utilized to transmit
cache operations to the system bus and indicate which sectors of the **cache** line

are targets of the **cache** operation.

TITLE - TI (1):
Multiprocessor computer system with sectored **cache** line system bus protocol
mechanism

Parent Case Text - PCTX (2):
The present invention is related to the following applications filed concurrently with this application, each of which is hereby incorporated: U.S.
patent application Ser. No. 09/753,057 entitled "Multi Computer System with
Sectored **Cache** Line Mechanism For Load and Store Operation " now pending; and
U.S. patent application Ser. No. 09/752,863 entitled "Multiprocessor Computer
System with Sectored **Cache** Line Mechanism for **Cache** Intervention ", now pending.

Brief Summary Text - BSTX (3):
The present invention generally relates to computer systems, specifically to
multiprocessor computer systems having caches which share memory and are thus
required to maintain **cache** coherency, and more particularly to an improved
method of maintaining **cache** coherency in a **cache** architecture having sectored
cache lines.

Brief Summary Text - BSTX (5):
The basic structure of a conventional multiprocessor computer system 10 is
shown in FIG. 1. Computer system 10 has several processing units, two of which
12a and 12b are depicted, which are connected to various peripheral devices,
including input/output (I/O) devices 14 (such as a display monitor, keyboard,
graphical pointer (mouse), and a permanent storage device or hard disk), memory
device 16 (such as random access memory or RAM) that is used by the processing
units to carry out program instructions, and firmware 18 whose primary purpose
is to seek out and load an operating system from one of the peripherals (usually the permanent memory device) whenever the computer is first turned on.
Processing units 12a and 12b communicate with the peripheral devices by various
means, including a generalized interconnect or bus 20, or direct memory

access
channels (not shown). Computer system 10 may have many additional components
which are not shown, such a serial, parallel, and universal system bus (**USB**)
ports for connection to, e.g., modems, printers or scanners. There are other
components that might be used in conjunction with those shown in the block
diagram of FIG. 1; for example, a display adapter might be used to control a
video display monitor, a memory controller can be used to access memory 16,
etc. The computer can also have more than two processing units.

Brief Summary Text - BSTX (6):

In a symmetric multi-processor (SMP) computer, all of the processing units
are generally identical, that is, they all use a common set or subset of
instructions and protocols to operate, and generally have the same
architecture. A typical architecture is shown in FIG. 1. A processing unit
includes a processor core 22 having a plurality of registers and execution
units, which carry out program instructions in order to operate the computer.
An exemplary processing unit includes the PowerPC.TM. processor marketed by
International Business Machines Corp. The processing unit can also have one or
more caches, such as an instruction **cache** 24 and a data **cache** 26, which are
implemented using high speed memory devices. Caches are commonly used to
temporarily store values that might be repeatedly accessed by a processor, in
order to speed up processing by avoiding the additional latency of loading the
values from memory 16. These caches are referred as "on-board" when they are
integrally packaged with the processor core on a single integrated chip 28.
Each **cache** is associated with a **cache** controller (not shown) that manages the
transfer of data and instructions between the processor core and the **cache**
memory.

Brief Summary Text - BSTX (7):

A processing unit can include additional caches, such as **cache** 30, which is
referred to as a level 2 (L2) **cache** since it supports the on-board (level 1)

caches 24 and 26. In other words, **cache** 30 acts as an intermediary between memory 16 and the on-board caches, and can store a much larger amount of information (instructions and data) than the on-board caches can, but at a longer access penalty. For example, **cache** 30 may be a chip having a storage capacity of 512 kilobytes, while the processor may be an IBM PowerPC.TM. 604-series processor having on-board caches with 64 kilobytes of total storage. **Cache** 30 is connected to bus 20, and all loading of information from memory 16 into processor core 22 must come through **cache** 30. Although FIG. 1 depicts only a two-level **cache hierarchy**, multi-level **cache hierarchies** can be provided where there are many levels (L3, L4, etc.) of serially connected caches.

Brief Summary Text - BSTX (8):

In a multi-level **cache**, if a copy of a value is in every level of the **cache**, **the cache hierarchy** is referred to as being "inclusive." It is not necessary, however, to keep a copy of each value in the lower levels, and an inclusivity bit field may be added to the caches to indicate whether or not the **cache** is inclusive. For example, a three-level **cache** structure might provide an L3 **cache** which was not inclusive, such that a value residing in the L2 **cache** might not be present in the L3 **cache**. In this example, if an L2 **cache** issues a read command for a value that is not present in any of the caches of that processing unit, it can be passed to that L2 **cache** without (necessarily) loading it into the L3 **cache**.

Brief Summary Text - BSTX (9):

In an SMP computer, it is important to provide a coherent memory system, that is, to cause write operations to each individual memory location to be serialized in some order for all processors. By way of example, assume a location in memory is modified by a sequence of write operations to take on the values: 1, 2, 3, 4. In a **cache** coherent system, all processors will observe

the writes to a given location to take place in the order shown. However, it is possible for a processing element to **miss** a write to the memory location. A given processing element reading the memory location could see the sequence 1, 3, 4, missing the update to the value 2. A system that implements these properties is said to be "coherent". Nearly all coherency protocols operate only to the granularity of the size of a **cache** block. That is to say, the coherency protocol controls the movement of and write permissions for operand data or instructions on a **cache** block basis, and not separately for each individual memory location.

Brief Summary Text - BSTX (10):

There are a number of protocols. and techniques for achieving **cache** coherence that are known to those skilled in the art. All of these mechanisms for maintaining coherency require that the protocols allow only one processor to have a "permission" that allows a write operation to a given memory location (**cache** block) at any given point in time. As a consequence of this requirement, whenever a processing element attempts to write to a memory location, it must first inform all other processing elements of its desire to write the location and receive permission from all other processing elements to carry out the write.

Brief Summary Text - BSTX (11):

To implement **cache** coherency in a system, the processors communicate over a common generalized interconnect (i.e., bus 20). The processors pass messages over the interconnect indicating their desire to read from or write to memory locations. When an operation is placed on the interconnect, all of the other processors "snoop" (monitor) this operation and decide if the state of their caches can allow the requested operation to proceed and, if so, under what conditions. There are several bus transactions that require snooping and follow-up action to honor the bus transactions and maintain memory coherency.
The snooping operation is triggered by the receipt of a qualified snoop request, generated by the assertion of certain bus signals.

Instruction
processing is interrupted only when a snoop **hit** occurs and the snoop state
machine determines that an additional **cache** snoop is required to resolve the
coherency of the offended sector.

Brief Summary Text - BSTX (12):
This communication is necessary because, in systems with caches, the most
recent valid copy of a given block of memory may have moved from the system
memory 16 to one or more of the caches in the system (as mentioned above). If
a processor (say 12a) attempts to access a memory location not present within
its **cache hierarchy**, the correct version of the block, which contains the
actual (current) value for the memory location, may either be in the system
memory 16 or in one of more of the caches in another processing unit, e.g.
processing unit 12b. If the correct version is in one or more of the other
caches in the system, it is necessary to obtain the correct value from the
cache(s) in the system instead of system memory.

Brief Summary Text - BSTX (13):
For example, consider a processor, say 12a, attempting to read a location in
memory. It first polls its own L1 **cache** (24 or 26). If the block is not
present in the L1 **cache**, the request is forwarded to the L2 **cache** (30). If the
block is not present in the L2 **cache**, the request is forwarded on to lower
cache levels, e.g., the L3 **cache**. If the block is not present in the lower
level caches, the request is then presented on the generalized interconnect
(20) to be serviced. Once an operation has been placed on the generalized
interconnect, all other processing unit snoop the operation and determine if
the block is present in their caches. If a given processing unit has the block
requested by processing unit in its L1 **cache**, and the value in that block is
modified, and any lower level caches also have copies of the block, then their
copies are stale, since the copy in the processor's **cache** is modified.
Therefore, when the lowest level **cache** (e.g., L3) of the processing unit snoops

the read operation, it will determine that the block requested is present and
modified in a higher level **cache**. When this occurs with an in-line **cache**
structure, the L3 **cache** places a message on the generalized interconnect
informing the processing unit that it must "retry" it's operation again at a
later time, because the actual value of the memory location is in the L1 **cache**
at the top of the memory **hierarchy** and must be retrieved to make it available
to service the read request of the initiating processing unit.

Brief Summary Text - BSTX (14):

Once the request from an initiating processing unit has been retried, the L3
cache begins a process to retrieve the modified value from the L1 **cache** and
make it available at the L3 **cache**, main memory or both, depending on the exact
details of the implementation. To retrieve the block from the higher level
caches, the L3 **cache** sends messages through the inter-**cache** connections to the
higher level caches, requesting that the block be retrieved. These messages
propagate up the processing unit **hierarchy** until they reach the L1 **cache** and
cause the block to be moved down the **hierarchy** to the lowest level (L3 or main
memory) to be able to service the request from the initiating processing unit.

Brief Summary Text - BSTX (15):

The initiating processing unit eventually re-presents the read request on
the generalized interconnect. At this point, however, the modified value has
been retrieved from the L1 **cache** of a processing unit and placed into system
memory, and the read request from the initiating processor will be satisfied.
The scenario just described is commonly referred to as a "snoop push". A read
request is snooped on the generalized interconnect which causes the processing
unit to "push" the block to the bottom of the **hierarchy** to satisfy the read
request made by the initiating processing unit.

Brief Summary Text - BSTX (16):

Thus, when a processor wishes to read or write a block, it must communicate that desire with the other processing units in the system in order to maintain **cache** coherence. To achieve this, the **cache** coherence protocol associates with each block in each level of the **cache hierarchy**, a status indicator indicating the current "state" of the block. The state information is used to allow certain optimizations in the coherency protocol that reduce message traffic on the generalized interconnect and the inter-**cache** connections. As one example of this mechanism, when a processing unit executes a read it receives a message indicating whether or not the read must be retried (i.e., reissued later). If the read operation is not retried, the message usually also includes information allowing the processing unit to determine if any other processing unit also has a still active copy of the block (this is accomplished by having the other lowest level caches give a "shared" or "not shared" indication for any read they do not retry). Therefore, a processing unit can determine whether any other processor in the system has a copy of the block. If no other processing unit has an active copy of the block, the reading processing unit marks the state of the block as "exclusive". If a block is marked exclusive it is permissible to allow the processing unit to later write to the block without first communicating with other processing units in the system because no other processing unit has a copy of the block. Therefore, it is possible for a processor to read or write a location without first communicating this intention onto the interconnection, but only where the coherency protocol has ensured that no other processor has an interest in the block.

Brief Summary Text - BSTX (17):

The foregoing **cache** coherency technique is implemented in a specific protocol referred to as "MESI." In this protocol, a **cache** block can be in one of four states, "M" (Modified), "E" (Exclusive), "S" (Shared) or "I" (Invalid). Under the MESI protocol, each **cache** entry (e.g., a 32-byte sector) has two additional bits which indicate the state of the entry, out of the four possible states. Depending upon the initial state of the entry and the type of

access
sought by the requesting processor, the state may be changed, and a particular
state is set for the entry in the requesting processor's **cache**. For example,
when a sector is in the Modified state, the addressed sector is valid only in
the **cache** having the modified sector, and the modified value has not been
written back to system memory. When a sector is Exclusive, it is present only
in the noted sector, and is consistent with system memory. If a sector is
Shared, it is valid in that **cache** and in at least one other **cache**, all of the
shared sectors being consistent with system memory. Finally, when a seoctor is
Invalid, it indicates that the addressed sector is not resident in the **cache**.

Brief Summary Text - BSTX (18):

A further improvement in accessing **cache** blocks can be achieved using the
cache coherency protocol. This improvement, referred to as "intervention,"
allows a **cache** having control over a memory block to provide the data in that
block directly to another **cache** requesting the value (for a read-type
operation), in other words, bypassing the need to write the data to system
memory and then have the requesting processor read it back again from memory.
Intervention can generally be performed only by a **cache** having the value in a
block whose state is Modified or Exclusive. In both of these states, there is
only one **cache** block that has a valid copy of the value, so it is a simple
matter to source (write) the value over the bus 20 without the necessity of
first writing it to system memory. The intervention procedure thus speeds up
processing by avoiding the longer process of writing to and reading from system
memory (which actually involves three bus operations and two memory
operations). This procedure not only results in better latency, but also
increased bus bandwidth.

Brief Summary Text - BSTX (19):

There are many variations of the MESI protocol. The tagged ("T") state is
used to identify a **cache** block which is inconsistent with system memory
(i.e.

modified) and is further responsible for writing the correct (current) value to
memory upon deallocation (or to pass on the tag to another **cache** block during
intervention). The T state can be used to a modified value, by marking one of
the sharing blocks as (temporarily) responsible for maintaining a valid copy of
the value. The recently read ("R") state can be used to allow intervention
when the value is unmodified but shared among many caches, so as to
conveniently mark a single one of the sharing caches as being responsible for
intervention. The hover ("H") state allows a **cache** line to maintain an address
in the directory even though the corresponding value in the **cache** entry array
is an invalid copy, so that it can snoop the correct value for its processing
unit if the value happens to be broadcast as part of an intervention between
the caches of two other processing units.

Brief Summary Text - BSTX (20):

While the foregoing techniques are very useful in facilitating shared use of
the system memory among the various caches, there are still several
inefficiencies in these designs, particularly for sectored caches. For
example, a 128-byte **cache** line may be divided into four 32-byte sectors, with
each of the four sectors containing valid data. If a store operation writes
new data to one of the sectors, the entire **cache** line must be invalidated, even
though three of the four sectors in the line are still valid. Thereafter, if
the processing unit associated with that **cache** issues a request to read one of
the three valid sectors, the entire **cache** line must be obtained from elsewhere
in the memory **hierarchy** (either from system memory, or from another **cache** via
intervention), even though the valid data is already present in the **cache**.
Although separate coherency bits could be provided for each of the sectors,
this approach would effectively remove the benefits that are otherwise achieved
from sectoring.

Brief Summary Text - BSTX (21):

In light of the foregoing, it would be desirable to devise an improved
method of handling **cache** transactions which did not require the

invalidation of
otherwise valid sectors in a **cache** line. It would be further advantageous if
the method could also provide for more efficient **cache** intervention.

Brief Summary Text - BSTX (23):
It is therefore one object of the present invention to provide an improved
method of maintaining **cache** coherency in a multiprocessor system.

Brief Summary Text - BSTX (24):
It is another object of the present invention to provide such a method which
does not require invalidation of portions of a sectored **cache** line when one
sector becomes invalid.

Brief Summary Text - BSTX (25):
It is yet another object of the present invention to provide such a method
which allows for partial intervention of requested data, that is, intervention
by transmitting less than the entire **cache** line.

Brief Summary Text - BSTX (26):
The foregoing objects are achieved in a method of maintaining coherency
among a plurality of caches each associated with a respective processing unit
of a multiprocessor computer system, wherein each of the caches has a plurality
of **cache** lines, and each of the **cache** lines is divided into a plurality of
sectors, the sectors having a smaller granularity than the **cache** lines, the
method generally comprising the steps of assigning a first **cache** coherency
state to a first sector of a **cache** line of one of the caches, and assigning a
second **cache** coherency state, different from the first **cache** coherency state,
to the **cache** line while maintaining the first **cache** coherency state for the
first sector of the **cache** line. For example, the first **cache** coherency state
may provide an indication that the first sector contains a valid value which is
not shared with any other **cache** (i.e., an exclusive or modified state). Also,
the second **cache** coherency state may provide an indication that at least one of

the sectors in the **cache** line contains a valid value which is shared with at least one other **cache** (a shared, recently-read, or tagged state). Other coherency states may be applied to other sectors in the same **cache** line. Partial intervention may be achieved by issuing a request to retrieve an entire **cache** line, and sourcing only a first sector of the **cache** line in response to the request. A second sector of the same **cache** line may be sourced from a third **cache**. Other sectors may also be sourced from a system memory device of the computer system as well. Appropriate system bus codes are utilized to transmit **cache** operations to the system bus and indicate which sectors of the **cache** line are targets of the **cache** operation.

Drawing Description Text - DRTX (4):

FIG. 2, comprises FIG. 2A and FIG. 2B, is a block diagram of one embodiment of a processing unit of a multiprocessor computer system constructed in accordance with the present invention, depicting the use of sectored **cache** lines having separate **cache** coherency states for each sector; and

Drawing Description Text - DRTX (5):

FIG. 3 is a timing diagram of various system bus signals which are utilized to implement a **cache** coherency protocol in accordance with the present invention wherein different coherency states can have different granularities.

Detailed Description Text - DETX (2):

With reference now to the figures, and in particular with reference to FIG. 2, there is depicted one embodiment 40 of a processing unit of a multiprocessor computer system constructed in accordance with the present invention. While the present invention is directed to a method of handling **cache** operations in a memory-coherent, multiprocessor system such as the system of FIG. 2, the present invention could be applied to computer systems that have additional hardware components not shown in FIG. 2, or having a different interconnection architecture (or both), so those skilled in the art will appreciate that the

present invention is not limited to the generalized system shown in that
figure.

Detailed Description Text - DETX (3):
Processing unit 40 is primarily comprised of a processor core 42 and L2
cache 44. Core 42 includes an L1 **cache** in this depiction, although it is not
shown separately. L2 **cache** 44 includes a 4-way set associative L2 directory
46, an L2 **cache** entry array 48, one or more snooper machines 50, and one or
more read/claim (RC) machines 52. Core 42 communicates with L2 **cache** 44 via
load and store ports 54 and 56, provided as part of the load/store unit (LSU)
of core 42. These address ports are provided as inputs to a multiplexor 58
whose output is fed to several tag comparators 60 used by L2 directory 46 to
determine whether an access request "hits" or "misses" **cache** 44 (i.e., whether
the requested address is contained in the L2 **cache**). In this example, 44 bits
of the address addr(20:63) are included in the output of multiplexor 58, but
only 28 bits addr(20:47) are fed to comparators 60. The other inputs to
comparators 60 come from the address tags of the four members of **index** line 62
in the 4-way set associative L2 directory 46 where each member's address tag
represents address(20:47) of the **cache** line residing in the L2 **cache** 48.

Detailed Description Text - DETX (4):
Each **index** line in the 4-way set associative L2 directory 46 corresponds to
four 128B **cache** lines in **cache** entry array 48, e.g., the four directory entries
in **index** line 62 corresponds to four 128B **cache** lines 64. The present
invention is directed to a **cache** architecture in which the **cache** lines are
sectored. **Cache** sectoring is generally known in the art, but the present
invention provides a novel approach to **cache** coherency for a sectored **cache**, as
explained further below. In the depicted embodiment, each **cache** line is 128
bytes long, with four sectors of 32 bytes each. Thus, each of the four **cache**
lines 64 has four sectors per **cache** line, and **index** line 62 has four

segments
for each of the four lines, corresponding respectively to each of the four
sectors of the four **cache** lines 64. Each input to comparators 60 from L2
directory 46 comes from a different one of these segments of **index** line 62.
The outputs of comparators 60 are combined in a logical OR fashion to indicate
a **hit**. When a **hit** is indicated, the particular comparator 60 having the high
output is used to identify the **hit** vector which points to the appropriate **cache**
line in the four 128B lines 64. This **hit** vector controls the selection for
another multiplexor 66 whose output provides the L2 **cache** value (operand data
or program instruction) to core 42. This **hit** vector also controls the
selection of multiplexer 66B which provides to RC 52 the state information for
each of the four 32B sectors of the 128B line **hit**.

Detailed Description Text - DETX (5):
In addition to examining the address information in L2 directory 46, it is
also necessary to examine coherency state information concerning each of the
sectors in order to determine if a valid copy of the requested value is present
in **cache** 46. It may be that a previous version of the value was stored in
cache 44, but an operation has overwritten that value in another processing
unit of the multiprocessor computer system, such that the old copy in L2 **cache**
entry array 48 is stale, that is, possibly incorrect. The present invention
uses **cache** coherency state such as the modified, exclusive, shared and invalid
states of the prior art MESI **cache** coherency protocol, with one important
difference--the present invention applies the various, coherency states to
different granularities depending upon the particular state.

Detailed Description Text - DETX (6):
In a preferred implementation of a **cache** coherency protocol according to the
present invention, the exclusive and modified states are applied only at the
sector (core) granularity (e.g., 32 bytes), while the invalid and shared states
may be applied at the overall **cache** line (memory) granularity (e.g.,

128
bytes), or at the sector granularity. Other coherency states may be used, such
as the "R" (recently read) state described in the Background section. The "R"
state may be applied to the overall **cache** line granularity as well.

Detailed Description Text - DETX (7):
Use of different granularities for different coherency states offers a
combination of benefits which have previously been unobtainable. For example,
in the simplest case wherein a particular requested value is present (valid) in
cache 44, it is not necessary that the other three sectors in that **cache** line
all be valid. In the prior art, a store operation from an adjacent processing
unit to a different sector in the same **cache** line would result in the
invalidation of the entire **cache** line. This result would thus require the
entire **cache** line to be retrieved from the remainder of the memory **hierarchy,**
even though the particular sector requested was actually valid in the **cache,**
thereby increasing latency. With the present invention, this inefficiency is
avoided. The state information for each of the four sectors of the **cache** line
is passed to RC machine 52 and, when a **hit** is indicated on a load operation,
the **hit** vector is used to check the coherency state for the particular sector.
If the state for that sector is valid (in the exemplary implementation,
exclusive, modified, recently read, or shared), then the L2 **cache** value is
passed to a core interface unit (CIU--not shown in core 42, regardless of the
states of the other sectors.

Detailed Description Text - DETX (8):
In addition to the coherency states that are assigned to each individual
sector, the overall **cache** line can have a coherency state as well, for the
above-noted states. For example, the overall **cache** line may be marked as
recently read ("R"), even though some sectors are invalid.

Detailed Description Text - DETX (9):
If the address requested by core 42 is not present (valid) in **cache** 44, the

value must be retrieved from elsewhere in the memory **hierarchy**, and loaded into **cache** 44. Retrieval is accomplished using RC. machine 52 which communicates with the system bus 68, as explained further below. If the block of memory corresponding to the **cache** line for the requested value is not present in **cache** 44, then a **cache** line in L2 **cache** entry array 48 must be made available. If the **cache** is full, a conventional technique may be used to evict a **cache** line, such as a least-recently used (LRU) algorithm. This invention uses a 4-way set associative directory in its example but it should be apparent to one skilled in the art that this invention could be easily applied to other directory structures (e.g. direct mapped directory).

Detailed Description Text - DETX (12):

Various commands from RC machine 52 to system bus 68 may be provided to support sector requests. In the event that a store operation is directed to a 128-byte line and all of the 32-byte sectors involved are already in L2 **cache** 44 in the modified state, then no system bus action is required at all (the store can be committed immediately to the L2 **cache**). For a DClaim operation, if any of the four sector bits indicate that a sector of the relevant line is shared, then RC machine 52 issues a system bus code for the DClaim operation and sets the req_sector_select(0:3) field to indicate which sectors the RC machine is requesting for the operation (one or more of these bits may be set). For a RWITM operation, if any of the four sector bits indicate that a sector is invalid, then RC machine 52 issues a system bus code for the RWITM operation and sets the req_sector_select(0:3) field to indicate which sectors the RC machine is requesting for the operation (one or more of these bits may again be set). For a load operation, RC machine 52 may ask for all 128 bytes, by setting all four of the req_sector_select(0:3) bits. Nevertheless, RC machine 52 may ask for a subset of the line, for any reason, such as a LARX (load-and-reserve) request involved in acquiring a lock variable, which usually has a granularity smaller than the full **cache** line. RC machine 52 can

also
perform a traditional load/RWITM/DClaim (of the full 128-byte line) by simply
turning on all of the sector select bits in the system bus command.

Detailed Description Text - DETX (13):
Each **cache** also includes a snooper 50 which is adapted to handle the 32-byte
system bus requests. Snooper machine 50 accepts a load/RWITM/DClaim request
from system bus 68 based on req _sector _select(0:3) which was sent by another
L2 master RC machine (i.e., of a processing unit other than processing unit
40). The control logic 108 of snoop machine 50 also includes a 4-bit sector
select register 110. Because a given snooper may only have a portion (e.g., 32
bytes) of the line being requested, that snooper may only be able to intervene
with that portion of the **cache** line. The snooper uses data _sector
_select(0:3) to identify which portion it is returning. It is therefore
possible for a master to request a load of all 128 bytes of a **cache** line and
have four different snoopers deliver the four different sectors that they
respectively have cached in a modified state.

Detailed Description Text - DETX (14):
Main memory can supply an incomplete portion of the line as well. For
example, a first sector of a line may be modified in a first **cache** where the
remaining sectors are invalid, and a second sector of the line may be modified
in a second **cache** where the remaining sectors are likewise invalid, with no
other caches having any valid copies of any of the sectors. In this scenario,
the first and second caches will intervene with their respective sectors, and
memory controller 106 will determine that the remaining two sectors need to be
sourced by main memory.

Claims Text - CLTX (1):
1. A method of maintaining **cache** coherency in a multiprocessor computer
system having a plurality of processing units, wherein each of the processing
units has at least one **cache,** each of the caches has a plurality of
cache

lines, and each of the **cache** lines is divided into a plurality of sectors,
comprising the steps of: assigning a first **cache** coherency state to a first
sector of a **cache** line of a first one of the caches; assigning a second **cache**
coherency state, different from the first **cache** coherency state, to the **cache**
line while maintaining the first **cache** coherency state for the first sector of
the **cache** line of the first **cache**; and issuing a system bus code from the
first **cache** to a system bus indicating that the first sector of the **cache** line
is a target of a **cache** operation of the first **cache**.

Claims Text - CLTX (2):
2. The method of claim 1 wherein said issuing step issues the system bus
code to further indicate that a second sector of the **cache** line is also a
target of the **cache** operation.

Claims Text - CLTX (3):
3. The method of claim 1 wherein: the first **cache** coherency state indicates
that the first sector of the **cache** line is shared with at least one other
cache; and said issuing step issues the system bus code to further identify
that the first sector of the **cache** line is a target of a DClaim operation.

Claims Text - CLTX (4):
4. The method of claim 1 wherein: the first **cache** coherency state indicates
that the first sector of the **cache** line is shared with at least one other
cache; and said issuing step issues the system bus code to further identify
that the first sector of the **cache** line is a target of a
read-with-intent-to-modify operation.

Claims Text - CLTX (5):
5. The method of claim 1 wherein said issuing step includes the step of
setting at least one bit in a sector select register of the **cache**, the set bit
being associated with the first sector of the **cache** line:.

Claims Text - CLTX (6):
6. The method of claim 5 wherein said setting step sets all of the bits in the sector select register of the **cache** to indicate that all of the sectors in the **cache** line are demanded.

Claims Text - CLTX (7):
7. The method of claim 1 further comprising the step of assigning a third **cache** coherency state to the first sector of the **cache** line in response to said issuing step.

Claims Text - CLTX (8):
8. A computer system comprising: a system memory device; a plurality of processing units each having a **cache**, each of said caches having a plurality of **cache** lines, and each of said **cache** lines being divided into a plurality of sectors; bus means for interconnecting said system memory device and said processing units; and **cache** coherency means for assigning a first **cache** coherency state to a first sector of a **cache** line of a first one of said caches, assigning a second **cache** coherency state, different from the first **cache** coherency state, to said **cache** line while maintaining the first **cache** coherency state for said first sector of said **cache** line, and issuing a system bus code from said first **cache** to said bus means indicating that said first sector of the **cache** line is a target of a **cache** operation of said first **cache**.

Claims Text - CLTX (9):
9. The computer system of claim 8 wherein said **cache** coherency means issues the system bus code to further indicate that a second sector of the **cache** line is also a target of the **cache** operation.

Claims Text - CLTX (10):
10. The computer system of claim 8 wherein: the first **cache** coherency state indicates that said first sector of said **cache** line is shared with at least one

other **cache**; and said **cache** coherency means issues the system bus code to
further identify that the first sector of said **cache** line is a target of a
DClaim operation.

Claims Text - CLTX (11):
11. The computer system of claim 8 wherein: the first **cache** coherency state
indicates that said first sector of said **cache** line is shared with at least one
other **cache**; and said **cache** coherency means issues the system bus code to
further identify that the first sector of said **cache** line is a target of a
read-with-intent-to-modify operation.

Claims Text - CLTX (12):
12. The computer system of claim 8 wherein said **cache** coherency means sets
at least one bit in a sector select register of said **cache**, the set bit being
associated with said first sector of said **cache** line.

Claims Text - CLTX (13):
13. The computer system of claim 12 wherein said **cache** coherency means sets
all of the bits in said sector select register of said **cache** to indicate that
all of the sectors in said **cache** line are demanded.

Claims Text - CLTX (14):
14. The computer system of claim 8 wherein **cache** coherency means further
assigns a third **cache** coherency state to said first sector of said **cache** line
in response to issuance of the system bus code.

on is present.

20 Claims, 6 Drawing figures

Exemplary Claim Number

US-PAT-NO: 6523092

DOCUMENT-IDENTIFIER: US 6523092 B1

TITLE: **Cache** line replacement policy enhancement to avoid
memory page thrashing

DATE-ISSUED: February 18, 2003

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Fanning; Blaise B.	El Dorado Hills	CA	N/A	N/A

US-CL-CURRENT: 711/134, 711/133

ABSTRACT:

A method for a **cache** line replacement policy enhancement to avoid memory
page thrashing. The method of one embodiment comprises comparing a memory
request address with **cache** tags to determine if any **cache** entry in set `n` can
match the address. The address is masked to determine if a thrash condition
exists. Allocation to set `n` is discouraged if a thrash condition is present.

20 Claims, 6 Drawing figures

Exemplary Claim Number: 1

Number of Drawing Sheets: 5

---------- KWIC ---------

Abstract Text - ABTX (1):
A method for a **cache** line replacement policy enhancement to avoid memory
page thrashing. The method of one embodiment comprises comparing a memory
request address with **cache** tags to determine if any **cache** entry in set `n` can
match the address. The address is masked to determine if a thrash condition
exists. Allocation to set `n` is discouraged if a thrash condition is present.

TITLE - TI (1):
Cache line replacement policy enhancement to avoid memory page thrashing

Brief Summary Text - BSTX (2):
The present invention relates generally to the field of computers and computer systems. More particularly, the present invention relates to a **cache** line replacement policy to avoid memory page thrashing.

Brief Summary Text - BSTX (5):
Computer systems have evolved to include memory **hierarchies** comprising various types of long term storage, main memory, and caches. However, as one moves down the down the memory **hierarchy** from caches to long term storage, device access times increase dramatically. An ideal solution is to have enough **cache** memory or fast main memory available to service the currently executing program. But in most systems, such memory is present in only limited amounts or the program demands more memory than is available.

Brief Summary Text - BSTX (6):
Caches are generally used to keep often used or recently used data close to or within the processor. The idea is that by storing recently used data in close proximity to the processor, the next time a memory request is made for that particular data, a long memory access to main memory or the hard disk drive is not necessary. When a computer starts up, the **cache** is empty. But over time, the **cache** continues to fill up until there are no longer any empty entries for new data. This is not a problem as long as invalid entries are available for replacement. But if all existing entries are valid, the **cache** replacement logic must delete valid entries to make room for incoming data.

Brief Summary Text - BSTX (7):
In an efficiently designed computer system, the chipset needs to return requested data to the processor with minimum latency in order to provide

maximum system performance. A number of factors can influence this latency, including overall system loading and the memory technology deployed. One factor that can become significant is the specific interaction between transactions that are closely related by the time that they are issued to the processor bus. When these transactions cause a conflict or "page **miss**" in the memory system, read latency seen by the processor can increase dramatically. The impact of these conflicts is often minimized by the inherent randomness of transaction requests. However, certain application access patterns can cause pathological cases where every access is impacted by these page **miss** cases.

Drawing Description Text - DRTX (3):
FIG. 1 is a block diagram of one embodiment of a system employing a **cache** line replacement policy enhancement to avoid memory page thrashing;

Drawing Description Text - DRTX (4):
FIG. 2 is a block diagram of a **cache** memory to employ one embodiment of the present invention;

Drawing Description Text - DRTX (6):
FIG. 3B illustrates a four-way set associative 128 Kbyte **cache**;

Detailed Description Text - DETX (2):
A method and apparatus for a **cache** line replacement policy enhancement to avoid memory page thrashing is disclosed. The embodiments described herein are described in the context of a microprocessor, but are not so limited. Although the following embodiments are described with reference to processors, other embodiments are applicable to other integrated circuits or logic devices. The same techniques and teachings of the present invention can easily be applied to other types of circuits or semiconductor devices that use caches or memory.

Detailed Description Text - DETX (4):
Referring now to FIG. 1, a computer system 100 is shown. System 100 includes a component, such as a memory controller hub 116, employing a **cache**

replacement mechanism 106 in accordance with the present invention, such as in the embodiment described herein. System 100 is representative of processing systems based on the Intel PENTIUM.RTM. Pro, PENTIUM II, PENTIUM III, Itanium.RTM. microprocessors available from Intel Corporation of Santa Clara, Calif., although other systems (including PCs having other microprocessors, engineering workstations, set-top boxes and the like) may also be used. In one embodiment, sample system 100 may be executing a version of the WINDOWS.TM. operating system available from Microsoft Corporation of Redmond, Wash., although other operating systems and graphical user interfaces, for example, may also be used. Thus, the present invention is not limited to any specific combination of hardware circuitry and software.

Detailed Description Text - DETX (6):

System 100 includes a memory 120. Memory 120 may be a dynamic random access memory (DRAM) device, a static random access memory (SRAM) device, flash memory device, or other memory device. Memory 120 may store instructions and/or data represented by data signals that may be executed by processor 102. A **cache** memory 104 can reside inside processor 102 that stores data signals stored in memory 120. Alternatively, in another embodiment, the **cache** memory such as external **cache** 108 can reside external to the processor.

Detailed Description Text - DETX (8):

A **cache** replacement mechanism 106 to avoid memory page thrashing also resides in MCH 116. Alternate embodiments of a **cache** replacement mechanism 106 can also be used in microcontrollers, embedded processors, graphics devices, DSPs, and other types of logic circuits.

Detailed Description Text - DETX (9):

System 100 uses a proprietary hub interface bus 122 to couple the MCH 116 to the I/O controller hub (ICH) 130. The ICH 130 provides direct connections to some I/O devices. Some examples are the audio controller, firmware hub (BIOS) 128, data storage 124, legacy I/O controller containing user input and

keyboard
interfaces, a serial expansion port such as Universal Serial Bus (**USB**), and a
network controller 134. The data storage device 124 can comprise a hard disk
drive, a floppy disk drive, a CD-ROM device, a flash memory device, or other
mass storage device.

Detailed Description Text - DETX (11):
Processor caches most often are arranged so that the lower bits of a request
address are used to **index** into the memory array and the upper bits are compared
against a **cache** tag to determine the relevance of the data in the **cache** to the
request being offered. As a result, requests whose addresses differ only in
these upper bits are assigned to the same line of the **cache** array, since the
lower bits are used as the **index**. Because this behavior can cause inefficient
use of the **cache**, many caches are divided into a number of smaller sets. These
sets diffuse the effect of pathological address patterns by creating several
locations where a given request can be stored, thus dramatically reducing the
probability of an addressing pathology. In such multi-set (or multi-way)
caching implementations, requests whose addresses **miss the cache** are typically
assigned to a **cache** set on a least-recently-used,
least-recently-allocated, or
random basis.

Detailed Description Text - DETX (12):
One form of a **cache** that is often used is known as a write-back **cache**. In a
writeback **cache** implementation, write operations offered by the processor core
are retained in the **cache** and written back to memory only when the **cache** line
needs to be recycled in order to service another **cache** transaction.

Detailed Description Text - DETX (13):
FIG. 2 is a block diagram of a **cache** memory 200 to employ one embodiment of
the present invention. The **cache** memory 200 of FIG. 2 is four-way set
associative with `n` sets. Each of the `n` sets consists of four separate
entries. **Cache** 200 comprises of Bank 0 210, Bank 1 220, Bank 2 230,

and Bank 3
240. Each bank provides a block to store an entry for each set. For example,
Bank 0 210 stores Entry 0 212, Bank 1 220 stores Entry 1 222, Bank 2 240 stores
Entry 2 242, and Bank 3 260 stores Entry 3 262.

Detailed Description Text - DETX (14):
When a memory access occurs, the **cache** logic receives a memory address 270.
The address 270 is divided into two fields: tag 272 and block offset 276. The
value in the tag field 272 is used to compare with the tags in a set when
searching for a match. The block offset field 276 is used to select the
desired data from the entry.

Detailed Description Text - DETX (15):
Each **cache** entry 260 comprises of three fields: valid 262, tag 264, and data
266. The valid field 262 indicates whether the present entry is currently in
use or not. The tag field 264 contains part of the address 270 and is used
during a **cache** search to determine whether an entry is the one desired. The
data field 266 contains the data for this entry.

Detailed Description Text - DETX (16):
During a memory access, the **cache** logic parses address 270 into the separate
fields 272 and 276. Each tag 272 is compared with the tag 264 of each valid
entry in all sets of the **cache**. If a match is found, then the data 266 is sent
to the processor. But if no match is found, then the **cache** 200 does not have
the desired data. After the data is retrieved from main memory, the data will
also be stored into the **cache** 200 for future reference. If one or more of the
cache sets contains an invalid/empty entry at the location pointed to by the
block offset 276, then the data is simply stored in that set. But if all the
sets have valid data at the location indicated by the block offset 276, then
the **cache** replacement policy determines which of the entries to discard. Two
strategies are that are often used in **cache** replacement algorithms are: (1)
random and (2) least recently used (LRU). To spread allocation

uniformly,
candidate entries are randomly selected in the random strategy. The
LRU
strategy attempts to reduce the chance of throwing out information that
will be
needed soon and records the accesses made to each entry. The set
selected for
replacement is the one that contains data that has been unused for the
longest
time.

Detailed Description Text - DETX (17):
However, if the data at a given block offset in all of the sets is
repeatedly used, then frequent replacements may happen. For instance,
in the
four-way set associate of this example, there are four **cache** sets. But
if
there are more than four addresses that are repeatedly accessed and the
data
are all stored at the same block offset, then the **cache** will
continually swap
the entries of that block offset in and out.

Detailed Description Text - DETX (19):
When an access to a particular bank maps to a different page than
the
previous access to that bank, a page **miss** occurs. The memory
controller must
retire the data that is currently loaded into the page array back into
the
memory core in order to make room in the page array for the new data.
The
latency of this operation in typical DRAM subsystems today is around 30
or 40
nanoseconds (ns). Since a typical DRAM access is currently around 100
ns, this
page **miss** can impact the latency of a particular transaction by up to
40%.

Detailed Description Text - DETX (23):
The memory controller maintains the openness, or state of pages.
Only one
page per bank can be opened at any one time. A page **miss** is defined as
an
attempt to access a page that is not presently open. When a page **miss**
occurs,
the memory controller has to open the desired page. If that particular
bank
already has an opened page, that page has to be closed first before the
new
page is opened. For a page read, the memory controller typically opens
a page,
accesses the data, and leaves the page open. A page **hit** occurs when

the memory
access is to an already open page and the memory controller does not have to
perform a open/close step. Page empty occurs when the memory access is to page
in a bank does not have any presently open page. The memory controller simply
opens the page without performing a close page.

Detailed Description Text - DETX (24):
When a pathological pattern, such as a stream of writes or the data accesses
is applied to a processor **cache** and chipset memory controller combination, the
regularity of the accesses combined with the LRU line allocation policy can
cause worst case memory behavior. Many of these patterns can be detected and
defeated through a modification to the **cache's** LRU policy. One such
modification detects likely page-thrashing conditions between reads and
writeback data, and modifies the line allocation policy in response.

Detailed Description Text - DETX (25):
For example, take the case of a 128 Kbyte processor **cache** that is divided
into four sets. FIG. 3B illustrates a four-way set associative 128 Kbyte
cache. Each set can store 32 Kbyte of data. Each **cache** line is in this case
32 bytes wide, although other line sizes are possible in other embodiments. If
the **cache** is empty and clean at initialization, the application of a 256 Kbyte
write stream to the **cache** would first cause 128 Kbyte of read/invalidate
operations. These read/invalidate or read-for-ownership (RFO) operations serve
to acquire ownership of the **cache** lines for the requesting agent. After these
first RFO operations, however, an undesirable behavior would begin in the
cache. The accesses that attempt to write to the **cache** beginning at 128 Kbyte
would cause an RFO operation from system address 128K. This RFO operation
would be followed by a writeback operation to system address 0.

Detailed Description Text - DETX (26):
In a minimal SDRAM system that employs four 64 megabit SDRAM devices,
addresses could map into the memory devices in such a way that accesses to
pages separated by 64 Kbyte would address the same memory bank. As a

result,
both accesses, the write to 128K and the writeback at 0, would be to the same
memory device. In fact, because of the system address-to-device bank mapping
discussed earlier, both accesses would be to the same bank. This causes a page
miss for the write operation. The memory controller needs to close the memory
page that is currently open in that bank and then open up the 128K page. The
next system transaction to be presented is likely be an RFO operation from 128
k+32 byte. This would again cause a page **miss** to occur because the last
address was to the same device in the same bank. These page misses would
continue through the entire 128 Kbyte of the **cache,** causing worst-case behavior
the whole time. Note that for this example, the read data for the 128K address
is destined for set 0 in the **cache** because of the LRU algorithm. Note also
that the dirty data in Sets 1 and 3 would not conflict with the address stream.
If the page thrash condition can be detected before the processor decides which
set's line should be replaced, then the pathological behavior described can be
avoided by the **cache** controller if the controller allocates the replacement
line from Set 1 or Set 3 instead. There are no interesting boundary cases
where all lines would cause page conflicts since correctness is not compromised
by the failure of this mechanism. If all the sets have conflicts and are
discouraged, then the **cache** controller can simply revert to the default **cache**
entry replacement scheme such as LRU.

Detailed Description Text - DETX (27):

In order to detect the thrash condition, the **cache** controller requires
information about the paging behavior of the memory subsystem. Sometimes this
information can be complex, as in the case where several different memory
technologies are populated in the same platform. The present invention
provides a mask that the **cache** controller can use to drive its decisions about
page conflicts.

Detailed Description Text - DETX (28):

FIG. 4 is a schematic diagram of a thrash condition detector. **Cache** controller 400 receives input signals THRASH BIT MASK 402 and REQUEST ADDRESS 404. THRASH BIT MASK 402 is a set of address mask signals coupled to AND gate 410 to detect the thrash condition. The mask 402 ensures that only the necessary bits of the addresses are tested. The particular configuration of the mask can be affected by the memory device type, device size, and bank size. REQUEST ADDRESS 404 is the address of the current memory access. SET `n` TAGS 406 are the **cache** tags of Set `n` to which the new data is to be stored. For this example, REQUEST ADDRESS 404 and each tag of SET `n` TAGS 406 both comprise of thirty two bits each.

Detailed Description Text - DETX (29):

Exclusive OR gates 408 receives REQUEST ADDRESS 404 and SET `n` TAGS. Exclusive OR gates 408 compares the inputs and generates a **hit or miss** result 409. The exclusive OR result 409 is gated by the THRASH BIT MASK 402 at AND gate 410. The detection result of the mask bits 402 and the tag comparison 409 is delivered at the AND gate 410 output as the DISCOURAGE `n` ALLOCATION signal 412. If the DISCOURAGE `n` ALLOCATION signal 412 is active, meaning that a page conflict may occur in the present set, then the **cache** controller 400 can attempt to allocate the replacement line from another set instead.

Detailed Description Text - DETX (30):

Embodiments of this mask can be used in the controller to determine whether a dirty entry is likely to conflict with the current request. If a possible conflict is detected and another set has a clean entry or a non-conflicting dirty entry, then that other set's entry is allocated instead. For the example above, the mask would consist of the bits that "bound" the conflicting addresses' ranges. Since accesses separated by 64 Kbyte cause page conflicts, but 32 Kbyte separations do not, bit 16 of the mask should be set while bit 15 remains clear. The mask starts with bit 0 as the right most bit. Furthermore, since a row of SDRAM implemented with four 64 megabit devices provides 32 megabytes of storage, accesses to addresses above 32 megabyte would be to a

different device and would necessarily not conflict with accesses to this
device. The bit mask derived from these conditions needs to include bit 16,
bit 24, and all bits in between in order for the **cache** controller to have
enough information to avoid page conflicts. Therefore, the mask for this
example would be 0x01FF0000 in hex.

Detailed Description Text - DETX (32):
The mask can be stored in the system BIOS or chipset and loaded into the
processor during system startup. If the processor reads this mask at startup
and stores the mask internally, then the mask can be provided to the **cache**
controllers to filter **cache** request addresses and to discourage allocation of a
line on a case-by-case basis.

Detailed Description Text - DETX (34):
FIG. 5 is a flow diagram for the method of one embodiment. At step 502, the
thrash mask is loaded into the **cache** controller. The address of the memory
request is also received at the controller at step 504. The tags for the set
`n` that can fulfill the memory request are received at the controller at step
506. At step 508, the set `n` tags are compared with the memory request
address to determine if there is a **cache hit**. If there is a **miss**, the address
is masked at step 510 to determine if a conflict can happen. If the controller
at step 512 determines that a conflict will not occur, then the check is
complete. But if a conflict is possible, then allocation to that set is
discouraged at step 514. The process continues indefinitely every time a new
memory address is received at step 504.

Claims Text - CLTX (1):
1. A method comprising: comparing a memory request address with **cache** tags
to determine if any **cache** entry in set `n` can match said address; masking
said address to determine if a thrash condition exists; and discouraging
allocation to set `n` if a thrash condition is present.

Claims Text - CLTX (4):
4. The method of claim 1 further comprising fulfilling said memory request
with a **cache** entry from a set other than set `n`.

Claims Text - CLTX (5):
5. The method of claim 1 further comprising defaulting to a normal **cache**
replacement scheme if said thrashing condition does not exist.

Claims Text - CLTX (6):
6. The method of claim 1 further comprising defaulting to a normal **cache**
replacement scheme if all **cache** sets are discouraged.

Claims Text - CLTX (7):
7. The method of claim 5 wherein said normal **cache** replacement scheme is a
least recently used scheme.

Claims Text - CLTX (8):
8. The method of claim 5 wherein said normal **cache** replacement scheme is a
least recently allocated scheme.

Claims Text - CLTX (9):
9. The method of claim 5 wherein said normal **cache** replacement scheme is a
random replacement scheme.

Claims Text - CLTX (10):
10. An apparatus comprising: a comparator circuit to compare a memory
request address with **cache** tags from set `n`, said comparator circuit to
determine whether a **cache** entry in said set `n` matches said memory request
address, said comparator circuit to output a memory address of any matched
entries; and a mask circuit coupled to said comparator circuit, said mask
circuit to mask an outputted memory address and to determine whether said
outputted memory address can cause a thrash condition.

Claims Text - CLTX (13):
13. The apparatus of claim 10 further comprising an input to receive said

cache tags from a **cache** memory.

Claims Text - CLTX (14):
14. The apparatus of claim 10 wherein said apparatus is a **cache** controller.

Claims Text - CLTX (16):
16. A system comprising: a processor coupled to a **cache**; a memory coupled to said **cache**; a **cache** controller coupled to said **cache** and said processor, said **cache** controller comprising: a comparator circuit to compare a memory request address with **cache** tags from set `n`, said comparator circuit to determine whether a **cache** entry in said set `n` matches said memory request address, said comparator circuit to output a memory address of any matched entries; and a mask circuit coupled to said comparator circuit, said mask circuit to mask an outputted memory address and to determine whether said outputted memory address can cause a thrash condition.

Claims Text - CLTX (17):
17. The system of claim 16 wherein said **cache** controller further comprises a latch to store a thrash bit mask, said latch coupled to said mask circuit and to supply said thrash bit mask to said mask circuit.

Claims Text - CLTX (18):
18. The system of claim 16 wherein said **cache** controller further comprises an input to receive said memory request address from said processor.

Claims Text - CLTX (19):
19. The system of claim 16 wherein said **cache** controller further comprises an input to receive said **cache** tags from said **cache**.

US-PAT-NO: 6622208

DOCUMENT-IDENTIFIER: US 6622208 B2

TITLE: System and methods using a system-on-a-chip with soft
cache

DATE-ISSUED: September 16, 2003

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
North; Gregory Allen	Austin	TX	N/A	N/A

US-CL-CURRENT: 711/118, 711/128 , 711/202 , 711/203 , 711/212 , 711/220
, 711/3

ABSTRACT:

A soft cache system compares tag bits of a virtual address with tag fields
of a plurality of soft cache register entries, each entry associated with an
index to a corresponding cache line in virtual memory. A cache line size for
the cache line is programmable. When the tag bits of the virtual address match
the tag field of one of the soft cache entries, the **index** from that entry is
selected for generating a physical address. The physical address is generated
using the selected **index** as an offset to a corresponding soft cache space in
memory.

18 Claims, 14 Drawing figures

Exemplary Claim Number: 1

Number of Drawing Sheets: 8

---------- KWIC ---------

Abstract Text - ABTX (1):
A soft cache system compares tag bits of a virtual address with tag fields
of a plurality of soft cache register entries, each entry associated with an

index to a corresponding cache line in virtual memory. A cache line size for
the cache line is programmable. When the tag bits of the virtual address match
the tag field of one of the soft cache entries, the **index** from that entry is
selected for generating a physical address. The physical address is generated
using the selected **index** as an offset to a corresponding soft cache space in
memory.

Brief Summary Text - BSTX (8):
According to the inventive concepts, a cache system is disclosed which
compares tag bits of a virtual address with tag fields of a plurality of soft
cache register entries, each entry associated with an **index** to a corresponding
cache line in a programmable cache memory space. A cache line size of the
cache line is also programmable. When the tag bits of the virtual address
match the tag field of one of the soft cache entries, the **index** from that entry
is selected for generating a physical address. The physical address is
generated using the selected **index** as an offset to a corresponding programmable
soft cache space in memory address by block address bits from the virtual
address.

Brief Summary Text - BSTX (9):
Systems and software embodying the principles of the present invention
advantageously allow software caching in systems with limited or no hardware
caching support. An example is the disclosed system-on-a-chip employing an
ARM7tdmi CPU core, which has neither a hardware cache nor a memory management
unit. Additionally, since cache line replacement is done in software on a
cache **miss**, the interface between memories operating under different protocols
has increased efficiency.

Drawing Description Text - DRTX (8):
FIG. 6 is a functional block diagram of the external **SRAM**/Flash memory
controller (SMC) of the system of FIG. 1;

Detailed Description Text - DETX (7):
Among the other devices operating directly off main AHB bus 103 are a 4-channel DMA engine 106, and flash/**SRAM** interface 107, including an external memory controller, which maps up to 512 MByte external memory into the microprocessor memory space as an extension of on-chip memory, a test interface controller (TIC) 108, arbiter 109 and LCD interface 110. Test Interface Controller (TIC) 108 can take over the bus control from microprocessor 101 and mimic the bus cycle in order to stimulate the blocks connected to AHB/APB buses. Arbiter 109 arbitrates bus requests on main bus 103. LCD interface 110 supports connections to various LCD panels (since the display may require a large frame buffer, display controller 110 operates from the high speed bus).

Detailed Description Text - DETX (50):
External **SRAM**/Flash Memory Controller (SMC) 107 supports eight external memory blocks, each having an address space up to 6 M Bytes. In the preferred embodiment of system 100, 4 SRAMs blocks are used, each with 1 M-byte address space, along with one block of FLASH RAM, as shown in FIG. 6. Each Bank has its own configuration register with which programmers can configure the Bank to support a specific type of External memory. In FIG. 6, the banks designated BANK 0-3 comprise 16-bit external SRAMs and the bank designated BANK4 comprises 8-bit wide external Flash RAMs.

Detailed Description Text - DETX (65):
USB port 114 can operate in either Configuration 0, where the control endpoint is for standard commands or Configuration 1, where the control endpoint is for Vendor/Class commands. String **Descriptor** Control Logic 1110 decodes string commands in the standard configuration. Vendor/Class commands are not decoded, but instead are stored and an interrupt to microprocessor 101 by Vendor/Class Control Logic. Subsequently, microprocessor 101, under software control, decodes Vendor/Class commands.

Detailed Description Text - DETX (68):
Each of the debug sub-block has a 1 bit read only "owner" field.

When this
bit is "0" the owner of this block is Microprocessor 101 and when this bit is
"1" the owner is DSP 102. The owner can write to the "Other Wr" field which,
if set, allows the other processor to write to the remaining fields of the
debug block. The "Debug En" field is set before the actions of the debug block
become effective and is also used to save power when this block is not in use.
The "Clk Freeze En" is a 1-bit field which when set at the trigger of a debug
event freezes the DSP clock. "Mem" is a 2 bit field which selects memories to
be observed for debug event. Start address range and end address range are
set, so that when the address is between these ranges an event triggers. The
"**Hit** Count" tracks the number of times a debug event has occurred.
Num_hits_for_trigger is the field which sets the number of hits to the address
range before the interrupt is enabled or clock is frozen.

Detailed Description Text - DETX (83):
Historically, embedded DSPs and uncached CPUs have either run code directly
out of external parallel memory (DRAM, **SRAM**, NOR FLASH) or out of internal
SRAM/FLASH. With respects to running code from external parallel memory, CPU
speed and power are adversely impacted. On the other hand, internal **SRAM**/FLASH
memory is typically significantly smaller thereby forcing direct memory
management during code development to insure proper paging. In particular, all
routines must fit within certain footprint guidelines to allow paging and
concurrent operation. In certain applications where all of the code is
developed in assembly language, this restriction can be addressed; however,
using a higher level languages like "C" for fast code development, the problem
is difficult, if not impossible, to solve.

Detailed Description Text - DETX (85):
As discussed further below in detail, the present soft cache system solves
these problems for low-performance code. Among other things, a hardware assist
is provided to manage existing generic on-chip **SRAM** as a cache. Additionally,
the combination of less than 1000 gates and a software abort handler provide a

means of managing code residing in NAND Flash as directly executable. Moreover, this cache mechanism is optimized for slow rate memories with lines
size programmable at 128, 256, 512, and 1kbytes per line and uses 16 way set
associative logic to provide 16 lines of cache space within the local **SRAM**.

Detailed Description Text - DETX (86):
The soft cache system preferably uses external **SRAM** for storing code and
on-chip memory for data constants or other secure information. In the illustrated embodiment, a virtual (soft cache) memory space of 2 Mbytes of
external and internal **SRAM** is dedicated to soft cache, although the size, as
well as the location in memory, are not critical and can change in actual
implementations.

Detailed Description Text - DETX (90):
The cache tag from the virtual address is compared with the tag fields in
the corresponding 16 register entries by the hardware comparators at Step 1403.
If a match occurs, the **index** corresponding to the matching entry in soft cache
is taken for use in generating a physical address to the appropriate location
in the soft cache memory block. For a 16-way cache system, the entries are
indexed with four bits from 0 to 15. In an embodiment where the soft cache
address space starts at 0000 and works upward, bits N-1:0 of the CPU address
become bits N-1:0 of the physical address and address the location in the cache
line. The 4-bit **index** from the matching soft cache entry replaces bits N+3:N
of the physical address and operates as an offset which selects the proper
cache line. The remaining bits (31:N+4) come from the virtual address and are
used for RAM bank, block and chip select. The required read or write to the
addressed area in the soft cache memory space subsequently takes place.

Detailed Description Text - DETX (91):
For example, assume that the cache line size is programmed to be 256 bytes.
In this case, N=8. Bits 21:8 of the CPU address are therefore compared with
the cache tags in the table entries. Also assume that a **hit** occurs to

entry #5
and the corresponding 4-bit **index** is 5. For simplicity, the soft cacheable
block of memory will have an arbitrarily selected starting address of 0. The
physical address (in hex) is therefore 000005CC, where CC are bits 7:0 of the
CPU address to a location within cache line 5.

Detailed Description Text - DETX (93):
Next consider the case where a cache **miss** occurs during comparison Step
1403. A hardware soft cache history register is maintained which tracks the
last four hits (matches). Preferably, this register is a shift register which
shifts in the 4-bit **index** from each matching entry, if that **index** differs from
the **index** of the previous matching **index**. In other words, if two consecutive
indices are the same, no shifting takes place such that no two consecutive
indices in the shift register are equal. This register improves the
performance of the software replacement handler invoked on a cache **miss**.

Detailed Description Text - DETX (94):
On the **miss**, a soft cache abort is signaled to microprocessor 101 by setting
an Abort Status bit in register. (This bit is cleared automatically after
being read). The address causing the cache **miss** (abort) is written into an
Abort Address register. Setting the Abort Status bit initiates the replacement
handler routine.

Detailed Description Text - DETX (95):
The replacement handler routine selects the cache line to be replaced based
on the contents of the history register. For example, the cache line replaced
can be one of those not corresponding to the four indices stored in the history
register. The required data is fetched from its current location in memory,
using the address in the Abort Address register and loaded into the selected
cache line, and the cache tag in the corresponding cache entry is updated. The
source location could be in NAND or serial flash and the software handler is
capable of performing the specific access procedures normally required

for
these types of memory. To implement this memory, the DMA engine and other
system resources are invoked as required. Once the data has been encached and
the cache tag updated, the instruction causing the cache **miss** can be successfully reissued.

Detailed Description Text - DETX (98):
One exemplary use of present soft **cache** is in digital audio player applications of system 100. For example, all low-performance code can run
directly thru the soft **cache** with only the high performance code (**USB**, Audio

Claims Text - CLTX (1):
1. A soft cache system performing the operations of: comparing tag bits of
a virtual address with tag bits programmed in a plurality of register entries,
each entry associated with an **index** to a cache line of programmable cache line
size in virtual cache space, the virtual cache space location in memory space
programmed by a block address portion of the virtual address and the cache line
size programmed by a field in the virtual address; when the tag bits of the
virtual address match the tag bits of one of the programmable register entries,
selecting the **index** corresponding to said entry for generating a physical
address; and generating the physical address using the selected **index** as an
offset to the cache line and the block address from the virtual memory as an
address to the virtual cache space; and when the tag bits of the virtual
address do not match the tag bits of one of the register entries: selecting a
cache line in the virtual cache space for encaching data associated with the
virtual address; retrieving the data associated with the virtual address from
a second area of memory using the virtual address; encaching the data retrieved from the second area of memory in the selected cache line in the
virtual cache space; and updating the tag bits in the register associated with
the selected cache line.

Claims Text - CLTX (7):
7. A method of data caching comprising the steps of: setting up an

M-way
set associative cache comprising a register having M number of corresponding
entries each storing a tag value and indexed to a cache line in a selected
cache area in memory; setting a cache line size for the cache lines, the cache
tag value selected as a function of the selected cache line size; generating a
virtual address including a tag field, a cache line address field addressing
locations within the cache line size and a block address field addressing the
cache area in memory; comparing the tag field of the virtual address with the
tag values in the cache; when the tag field of the virtual address matches a
tag value in the cache, generating a physical address to the corresponding
cache line with the **index** associated with the register entry containing the
matching tag value and the cache line and block address fields from the virtual
address; updating a history register with the **index** associated with the
matching tag value; and when the tag field of the virtual address does not
match a tag value in the register, accessing a corresponding area in memory
with the virtual address; selecting a replacement cache line from the cache
area in memory from information stored in the history register; transferring
data from the corresponding area in memory accessed with the virtual address to
the replacement cache line in the cache area of memory; and updating the
register entry associated with the **index** to the replacement cache line with the
tag from the virtual address.

Claims Text - CLTX (8):
8. The method of claim 7 wherein said step of updating the history register
comprises the step of shifting an **index** into a shift register.

Claims Text - CLTX (14):
14. A system comprising: a central processing unit; a memory including a
soft cache space; and a soft cache system for accessing data in the soft cache
space at a programmable location within memory space in response to addresses
from the central processing unit including: a register having a

plurality of
entries each for storing a cache tag and indexed to a cache line in the soft
cache space, the cache line having a programmable cache line size; a plurality
of comparators for comparing a tag field of an address from the central
processing unit with cache tags stored in each of the entries of the register;
circuitry for generating an address to the soft cache space when the tag field
matches a cache tag in the register using the **index** associated with the entry
storing the matching cache tag and selected bits from the address from the
central processing unit addressing the soft cache space; and circuitry for
updating data within the soft cache space when the tag field does not match the
cache tag in the register and operable to select a cache line in the soft cache
space for encaching data associated with the address from the central
processing unit, retrieve the data from a second area of the memory space using
the address from the central processing unit, encache the data from the second
area of the memory space in the selected cache line, and update the cache tag
in the register associated with the selected cache line.